University of California
Santa Cruz

Design and Analysis of Distributed Routing Algorithms

A thesis submitted in partial satisfaction
of the requirements for the degree of
Master of Science
in
Computer Engineering
by
Shree N. Murthy
June 1994

The thesis of Shree N. Murthy is approved:

Prof. J. J. Garcia-Luna-Aceves

Prof. Anujan Varma

Prof. Darrell D. E. Long

Dean of Graduate Studies and Research

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **JUN 1994**	2. REPORT TYPE	3. DATES COVERED **00-06-1994 to 00-06-1994**
4. TITLE AND SUBTITLE **Design and Analysis of Distributed Routing Algorithms**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**69**	

Contents

List of Figures

List of Tables

Design and Analysis of Distributed Routing Algorithms

Shree N. Murthy

Abstract

Route assignment is one of the operational problems of a communication network, and adaptive routing schemes are required to achieve real-time performance. This thesis introduces, verifies and analyzes two new distributed, shortest-path routing algorithms, which are called, Path-Finding Algorithm (PFA) and Loop-Free Path-Finding Algorithm (LPA). Both algorithms require each routing node to know only the distance and the second-to-last-hop (or predecessor) node to each destination. In addition to the above information, LPA uses an efficient inter-neighbor coordination mechanism spanning over a single hop. PFA reduces the formation of temporary loops significantly, while LPA achieves loop-freedom at every instant by eliminating temporary loops. The average performance of these two algorithms is compared with the *Diffusing Update Algorithm* (DUAL) and an ideal link state (ILS) using Dijkstra's shortest-path algorithm by simulation; this performance comparison is made in terms of time taken for convergence, number of packets exchanged and the total number of operations required for convergence by each of the algorithms. The simulations were performed using a C-based simulation tool called *Drama*, along with a network simulation library. The results indicate that the performance of PFA is comparable to that of DUAL and ILS and that a significant improvement in performance can be achieved with LPA over DUAL and ILS.

Acknowledgments

I sincerely thank my advisor, J.J. Garcia-Luna-Aceves, for his invaluable guidance and constant encouragement. It was fun to work with him.

Thanks to Anujan Varma and Darrell D. E. Long for being in my Reading Committee and for giving incisive comments and insightful suggestions. Special thanks to Bill Zaumen for bearing with all my questions and patiently helping me out with *Drama*. Thanks also to Paul for helping with the installation of *Drama*.

Thanks to people in Concurrent Systems Lab for providing an excellent working environment. Thanks to Michelle for providing the format file for this thesis. Thanks also to all my friends for their support and encouragement.

Beyond all the people in my immediate environment, I am very thankful to the support, encouragement and numerous sacrifices of my parents.

This work was supported by the Office of Naval Research under Contract No. N-00014-92-J-1807.

Chapter 1
Introduction

A very important component of a network is the communication subnetwork. This includes the hardware and the software required for the transmission of data from one node to another. There are mainly two switching schemes – circuit-switching and packet-switching, which can be used for data transfer. The bursty nature of computer traffic favors packet-switched mode of transmission.

In a packet-switched network, messages are partitioned into packets, which are then transmitted through the network using store-and-forward switching. The selection of next hop towards a destination in a packet-switched network is made by a well defined decision rule referred to as the *routing policy*. Routing algorithms are referred to as the network layer protocol that guides packets through the communication subnet to their correct destination.

Routing policies can be classified as *deterministic* and *adaptive* depending on whether the routes change in response to the traffic input pattern. In a deterministic policy, the path a packet takes from a source i to a destination j is predetermined. In an adaptive policy, packets are routed such that the congested and damaged areas in the network are avoided. In other words, adaptive routing policies are adaptable to load fluctuations and changes in the topology of the network. The information maintained at each routing node or router is updated depending on the state of the network.

Adaptive routing algorithms can also be classified as *centralized* or *distributed*, depending on the way in which routing paths are computed. In a centralized approach, the path information is computed at one centralized node, whereas in a distributed approach, path information is computed at each routing node.

Distributed routing algorithms can be further classified into *distance-vector algorithms* (DVA) and *link-state algorithms* (LSA), depending on the method adopted to maintain routing information in router databases. In a DVA, a router knows the cost of the preferred path through each of its neighbors to all destinations and uses this information to compute the shortest path and the next node (successor) in the path to each destination. Each update message sent by a router to its neighbors contains a vector with one or more entries, each of which specifies as a minimum, the distance to a given destination. In a LSA, a node must receive information about the entire network topology to compute the preferred path to each destination. Each node broadcasts update messages containing the state of a node's adjacent links to every other node in the network. In this thesis, we concentrate on distance-vector algorithms for updating routing information maintained at each router (node).

Routing in today's computer networks is accomplished by distributed shortest-path routing algorithms. The routing algorithms based on Distributed Bellman-Ford (DBF) algorithm [BG92] are susceptible to the formation of temporary loops. Looping problems can be avoided in one of the three ways. OSPF [Moy91] relies on broadcasting complete topology information among routers, and organizes an internet hierarchically to cope with the overhead incurred with any topology broadcast algorithm. BGP [RL94] exchanges distance vectors that specify complete paths to destinations. Cisco's EIGRP uses a loop-free routing algorithm based on internodal coordination called DUAL [GLA93]

Recently, a number of distributed shortest-path routing algorithms have been proposed [CRKGLA89, GLA86, Hag83, Hum91, RF91] have proposed distributed shortest-path algorithms that utilize information regarding the length and the second-to-last hop (predecessor) of the shortest-path to each destination to eliminate the counting-to-infinity problem. We refer to these algorithms as *path-finding algorithms.* Path-finding algorithms are attractive alternative to DBF for distributed routing as they eliminate the *counting-to-*

infinity problem. However these path finding algorithms can incur substantial temporary loops in the paths specified by the predecessor information before they converge, which leads to slower convergence.

In this thesis, we introduce, verify and analyze two new shortest-path routing algorithms, which we call path-finding algorithm (PFA) and loop-free path-finding algorithm (LPA). Both of these algorithms operate by specifying the second-to-last-hop (or predecessor) to each destination, in addition to the distance to destination. Predecessor information is used to derive an implicit path to the destination without additional overhead. Any router can traverse the path specified by a predecessor from any destination back to its neighbor router to determine whether by using that neighbor as its successor would create a path that contains a loop (i.e, involves the router itself). Unlike earlier path-finding algorithms [CRKGLA89, Hum91], in PFA and LPA, upon receiving an update message from its neighbor k, node i also determines if a path to destination j through any of its other neighbors ($\neq k$) includes node k itself. If so, such a path is not selected. This step *reduces* the possibility of temporary loops. LPA achieves loop-freedom at every instant using the implicit path information and an inter-neighbor coordination mechanism that spans over single hop only.

PFA and LPA use the same amount of information as previous path-finding algorithms and have lower time and communication complexities. The algorithms can be made adaptive to the network load by choosing a proper *cost-metric* for updating routing information at each node. The results presented here can be used to develop a new implementation of the *Routing Information Protocol* (RIP) [Hed88] that eliminates all of its performance problems.

The rest of this thesis is organized as follows. Chapter 2 gives a brief overview about the development of routing algorithms and the state of the art of routing protocols. Chapter 3 describes and verifies the first of the two proposed algorithms, PFA. Chapter 4 describes and verifies LPA. Chapter 5 compares the performance of PFA and LPA with the performance of DUAL and ILS for a single node/link failure and addition; this comparison is made by simulations using a C-based simulation language *Drama* along with a network simulation library. Finally, Chapter 6 concludes with a brief summary and an insight into future work.

Chapter 2 Overview

Routing techniques for packet-switched networks can be broadly classified into *static* and *adaptive* routing policies [Sch86]. In static routing, routing tables are set up at a certain time before the data are being transmitted and the routing table is not changed thereafter. In adaptive routing, network conditions are continuously monitored and the routing tables are changed dynamically to adapt to the changing network conditions. Adaptive routing can be further subdivided into *centralized* and *distributed* routing, depending on the storage of the routing information. Henceforth, we refer to *adaptive, distributed* routing simply as routing.

At a fundamental level, for routing packets, a switch has to decide on which outbound queue to place a packet based on the destination address of the packet and quality of service (QoS) parameters. Routing protocols are responsible for forwarding the data packets over routes that provide good or optimal performance. Consequently, a routing protocol is required to maintain the status of all the routes in the network.

A routing node (or router) running a routing algorithm mainly consists of two parts – an initialization step and a step that is repeated until the algorithm terminates. The initialization step involves initializing all the routers in the network. The subsequent step involves updating minimum distance of each router for all destinations until the algorithm converges to correct distances. The routing algorithms differ in the way by which the updating step is implemented. There are two kinds of adaptive routing algorithms — *link* *state* and *distance vector* algorithms. In this thesis, we focus on shortest-path routing algorithms based on distance vectors.

Link-state Algorithms: In the link-state approach, each router[1] maintains a complete view of the network topology with a cost for each link. A router broadcasts regularly the link cost information of all its outgoing links to all other routers. Typically, this is done by flooding. That is, a router sends link cost information to all its neighboring routers, who in turn forward the same information to their neighbors and so on. When a router receives information about the change in a link cost, it updates its view of the network topology and applies a shortest path algorithm to choose its next hop for each destination.

Routers may not always have a consistent view of the network topology, because of the time updates take to reach all routers. This inconsistent view of the network can lead to the formation of loops, which are temporary and disappear in the time it takes for all routers to have the same topological information.

Shortest Path First (SPF) [McQ74] is a link-state protocol in which each node computes and broadcasts the costs of its outgoing links periodically and applies Dijkstra's shortest path algorithm [BG92] to determine the next hop; other routing protocols that work on the same link-state approach are IS-IS [Ora90, Per91], and OSPF [Moy91].

Distance-Vector Algorithms: In a distance-vector algorithm, a router knows the length of the shortest-path (distance) from each of its neighbors to every destination in the network, and uses this information to compute its own distance and next router (successor) to each destination. Well-known examples of routing protocols based on distance-vector algorithms, which we call distance vector algorithms (DVA), are the routing information protocol (RIP) [Hed88], the HELLO protocol [Mil83a], the gateway-to-gateway protocol (GGP) [HS82], the exterior gateway protocol (EGP) [Mil83b] and the old ARPANET routing protocol [McQ74]. All these DVAs have used variants of the distributed Bellman-Ford algorithm (DBF) for shortest-path computation [BG92]. The primary disadvantage of DBF

[1]we use 'router' and 'node' interchangeably

are *routing-table loops* and *counting-to-infinity* [GLA89]. A routing-table loop is a path specified in the routers' routing tables at a particular point in time, such that the path visits the same router more than once before reaching the intended destination. A router is said to be counting-to-infinity when it increments its distance to a destination until it reaches a predefined maximum distance value.

Because of the poor performance of DVAs implemented using DBF, DVAs were not considered to be a viable approach to supporting routing in large networks or internets. Recently, however, a number of efficient distance-vector algorithms have been proposed to eliminate the counting-to-infinity problem and routing-table loops [JM82, GLA92, GLA89]. In this thesis, we focus on the distance-vector algorithms which achieve loop freedom by making use of predecessor information.

2.1 Evolution of Distance-Vector Algorithms

One of the earliest implementations of DVA was the routing protocol implemented in the ARPANET in the early 1970s. In this protocol, every router in the network maintains a distance and a routing table. The shortest path information for each destination is maintained in node's routing table. Every node broadcasts its routing table information periodically to its neighboring nodes. A router examines its routing table to determine the shortest path to a particular destination before sending a packet to that destination.

One of the basic problems with this type of routing algorithm is the *counting-to-infinity* problem, in which a node counts to a maximum value (infinity) before converging after a node failed or a network partition. Many approaches have been proposed in the past to solve, at least in part, the looping problems of DVAs. A widely known proposal is the *split-horizon technique*, which avoids ping-pong looping, whereby two nodes choose each other as the successor to a destination [Ceg75, Sch86]. Another well known technique which has been proposed is the use of hold downs. Both of these approaches do not completely solve the counting-to-infinity problem [GLA89]. Some other solutions have also been proposed to overcome this problem [GLA89]. The routing algorithms discussed in this thesis eliminate the counting-to-infinity problem by maintaining the information about the second-to-last-hop (or predecessor) node as a part of the path information to each destination.

The periodic updates sent by a node to all its neighboring nodes can be of two types – *event driven* or *timer-driven*. In event driven routing, updates are sent when a certain event occurs. Typical events are the changing of a local metric value, or the reception of a routing update from a neighbor. In timer driven routing, updates are sent periodically, whether there is any change in the status of the network from the last update sent or not. Typically, updates are sent immediately after an event and when timer expires if no event occurs. For ease of exposition, we opt for event-driven routing updates in all our routing algorithms.

2.2 Network Model

A computer network G is modeled as an undirected graph represented as $G(V, E)$, where V is the set of nodes and E is the set of edges or links connecting nodes. Each node represents a router and is a computing unit involving a processor, local memory, and input and output queues with unlimited capacity. Extending the model to account for end node (link) destinations attached to routers is trivial. A functional bidirectional link connecting nodes i and j is denoted by (i, j) and is assigned a positive weight in each direction. A link is assumed to exist in both directions at the same time. All messages received (transmitted) by a node are put in the input (output) queue on a first-come-first-serve basis and are processed in that order. An underlying protocol assures that:

- Every node knows who its neighbors are; this implies that a node within a finite time detects the existence of a new neighbor or the loss of connectivity with a neighbor.
- All packets transmitted over an operational link are received correctly and in the proper sequence within a finite time.
- All update messages, changes in the link-cost, link failures and link recoveries are processed one at a time in the order in which they occur.

Each node is given a unique identifier. Any link cost can vary in time but is always positive. The distance between the two nodes in the network is measured as the sum of the link costs of the shortest path between nodes.

When a link fails, the corresponding distance entry in a node's distance and routing tables are marked as infinity. A node failure is modeled as all links incident on that node failing at the same time. A change in the operational status of a link or a node is assumed to be notified to its neighboring nodes within a finite time. These services are assumed to be reliable and are provided by the lower level protocols.

2.3 Notations and Definitions

A *path* from node i to node j is a sequence of nodes where (i, n_1), (n_x, n_{x+1}), (n_r, j) are links in the path. A *simple path* from i to j is a sequence of nodes in which no node is visited more than once. A *implicit path* from i to j is a path that is derived from predecessor node information. The paths between any pair of nodes and their corresponding distances change over time in a dynamic network. At any point in time, node i is connected to node j if a path exists from i to j at that time. The network is said to be connected if every pair of operational nodes are connected at a given time.

Throughout the paper the following notation is used:

j:	Destination node identifier $j \in N$
b, k:	Neighbor nodes
D^i_{jk}:	Distance entry at node i to destination j through neighbor k in the distance table
D^i_j:	Distance entry at node i to destination j in the routing table
p^i_{jk}:	Predecessor entry from i to j through k in the distance table
p^i_j:	Predecessor entry from i to j in the routing table
s^i_j:	Successor from node i to j
d_{ik}:	Link cost from i to neighbor k

N_i:	Set of neighbors of i
$FD^i_j(t)$:	Distance value used by node i to evaluate feasibility at time t
r^i_{jk}:	Reply status flag for a query sent by node i for j through k
$S_j(t)$:	Successor graph of G at i for destination j at time t
$C_j(t)$:	Loop formed for destination j at time t
$P_{xj}(t)$:	Path from node x to node j implied by successor entries at time t
$RD^i_j(t)$:	Distance from node i to node j at time t
$RH^i_j(t)$:	Predecessor of node j along the path from i to j at time t
$u^i_j(t)$:	Update flag
$tag^i_j(t)$:	Tag at node i for destination j at time t
$H(I,d)$:	Maximum number of links in the loop-free path from node i having a length not exceeding d in the final topology
$T(i)$:	Time by which all messages that are in transit at time $T(i-1)$ have reached the destination

The time at which the value of a variable applies is specified only when it is necessary. The successor to destination j for any node is simply referred to as the successor of that node, and the same reference applies to other information maintained by a node. Similarly, updates, queries and responses refer to destination j, unless stated otherwise.

In the algorithm's description, the time at which the value of a variable X of the algorithm applies is specified only when it is necessary; the value of X at time t is denoted by $X(t)$.

The next two chapters describe the two algorithms, PFA and LPA, and provide their pseudocode and formal proofs for correctness and complexity.

Chapter 3
A Path-Finding Algorithm

In this chapter, we present a new algorithm, called *path-finding algorithm* (PFA), which substantially reduces the number of cases in which routing loops can occur. A formal proof of PFA is presented and the worst-case complexity of the algorithm is analyzed.

The rest of this Chapter is organized as follows. The next section describes the operation of the algorithm with an example. Section 3.2 gives the formal proof for correctness of PFA. Section 3.3 discusses the time and communicational complexity of PFA. Section 3.4 compares PFA qualitatively with Humblet's path-finding algorithm. Finally, Section 3.5 concludes the Chapter with a brief summary.

3.1 PFA Description

PFA is specified in pseudocode form in Figure 3.1. The main feature of PFA is the notion of second to last hop or *predecessor*. Using predecessor information, each node can infer the path implicit in a distance entry without excessive overhead.

Each node maintains a *distance table* and a *routing table*. The distance table is a matrix containing the distance and predecessor entries (path information) for each destination through all the neighboring nodes. The routing table is a column vector containing the minimum distance entry for each destination, and its corresponding *predecessor*, and *successor* nodes (this can be extracted from the distance table). An update message mainly contains the source and the destination node identifiers, and the distance and predecessor for one or more destinations.

When a node i receives an update message from its neighbor k regarding destination j, the distance and predecessor entries in the distance table are updated (Step 1). A unique feature of PFA is that node i also determines if the path to destination j through any of its other neighbors $\{b \in N_i | b \neq k\}$ includes node k. If the path implied by the predecessor information reported by node b includes node k, then the distance entry of that path is also updated as $D^i_{jb} = D^i_{kb} + D^k_j$ and the predecessor is updated as $p^i_{jb} = p^k_j$. Thus, a node can determine whether or not an update received from k affects its other distance and routing table entries. Before updating the routing table, node i checks for all simple paths to j reported by its neighbors, and the shortest of these simple paths becomes the path from i to j (Procedure RT_Update). This implies that, at each stage, node i checks for the simple paths and avoids loops. Link or node failures, recoveries, and link-cost changes are handled similarly (Steps (5), (6) and (7)).

In contrast to PFA, which makes node i check the consistency of predecessor information reported by *all* its neighbors each time it processes an event involving a neighbor k, all previous path-finding algorithms [CRKGLA89, CKGLA92, RF91, Hum91] check the consistency of predecessor only for the neighbor associated with the input event. This unique feature of PFA accounts for its fast convergence after a single link-cost change, as illustrated in Section 5, because it eliminates more temporary looping situations than previous path-finding algorithms or even BGP [RL94] could.

The following example helps illustrate this. Consider the four-node network shown in Figure 3.1(a). Assume that PFA is used in this network and that all links and nodes have the same propagation delays. The costs of the links are as indicated in the figure; links have the same cost in both directions. Here, j is the destination node, k and b are neighbors of node i. The arrows next to links indicate the direction of update messages, the label in parentheses gives the distance to the destination and the predecessor to the destination j. The figure focuses on the update messages regarding destination j only.

When link (j, k) fails, nodes j and k send update messages to the neighboring nodes as shown in Figure 3.1(b). In this example, node k is forced to report an infinite distance

```
Procedure Update($(i, k)$)
when router $i$ receives an update on link $(i, k)$
(0) begin
        update=0; $RTEMP^i \leftarrow \phi$;
        $DTEMP^{i,b} \leftarrow \phi$ for all neighbors $b$
(1)     for each triplet $(j, D^k_j, p^k_j)$ in $V^{k,i}, j \neq i$ do
        begin
                $D^i_{jk} \leftarrow d_{ik} + D^k_j$; $p^i_{jk} \leftarrow p^k_j$;
(2)             for all neighbors $b$
                    if k is in the path from i to j in
                      the distance table through neighbor $b$
                    then $D^i_{jb} \leftarrow D^i_{kb} + D^k_j$; $p^i_{jb} \leftarrow p^k_j$
        end
(3)     begin
            if there are $b$ and $j$ such that
              $(D^i_{jb} < D^i_j)$ or $((D^i_{jb} > D^i_j)$ and $(b = s^i_j))$;
              then Call RT_Update;
        end
(4)     begin
            if $(RTEMP^i \neq \phi)$ then
              for each neighbor $b$ do
              begin
                      for each triplet $t = (j, D^i_j, p^i_j)$ in $RTEMP^i$ do
                          if $b$ is not in the path from $i$ to $j$
                            then $DTEMP^{i,b} \leftarrow DTEMP^{i,b} \cup t$;
                      send $DTEMP^{i,b}$ to neighbor $b$;
              end
        end
end

Procedure Change($(i, k, d_{ik})$)
when $d_{ik}$ changes value do
(7) begin
        update the distance table entry at node $i$
            $D^i_{jk} \leftarrow d_{ik} + D^k_j$; $p^i_{jk} \leftarrow p^k_j$;
        Go to Step (2);
end

Procedure Failure($(i, k)$)
when link $(i, k)$ fails do
(5) begin
        delete column $k$ in $D_i$
        if there is a destination $j$ such that $k = s^i_j$
        then Call RT_Update;
        Go to Step (4);
end
```

```
Procedure Recover($(i, k, d_{ik})$)
when link $(i, k)$ comes up do
(6) begin
        insert column $k$ in $D_i$
        respond as if a single entry in $V^{k,i} = (k, d_{ik}, i)$
        is received on link $(i, k)$
        copy whole routing table into $DTEMP^{i,k}$ and
        send it to $k$
end

Function In_Path(Node,Neighbor,Dest, neigh)
begin
        $p \leftarrow p^{Node}_{Dest,neigh}$;
        if $(p =$ Node$)$ then return(false);
        else if $(p =$ Neighbor$)$ then return(true);
             else In_Path(Node,Neighbor,p,neigh);
end

Procedure RT_Update
begin
    initialize all destinations to be unmarked;
    for any unmarked destination $j$ do
    begin
        if there is no finite distance in row $j$
        then mark $j$ as undetermined;
        else begin
                $TV \leftarrow \phi$;
                pick any minimum distance $D^i_{jb}$
                $c \leftarrow p^i_{jb}$, $TV \leftarrow TV \cup c$;
                repeat $c \leftarrow p^i_{cb}$, $TV \leftarrow TV \cup c$;
                    until $D^i_{cb}$ is not minimum of row
                    $c$ or $p^i_{cb} = i$ or $p^i_{cb}$ is marked
                if $((p^i_{cb}$ is marked as undetermined$)$ or
                  $(D^i_{cb}$ is not minimum of row $c))$
                then mark each node in $TV$ as undetermined
                else begin
                        mark each node in $TV$ as determined
                        $D^i_j \leftarrow D^i_{jb}$; $p^i_j \leftarrow p^i_{jb}$; $s^i_j \leftarrow b$;
                        end
                end
    end
    copy the routing vector to $RTEMP^i$ if the distance or
    predecessor has changed
end
```

FIGURE 3.1: PFA Specification

to j, because nodes b and i have reported node k as part of their paths to destination j. Figure 3.1(c) shows that node b processes node k's update and selects link (b, j) to destination j; This is the case because of step(2), which forces node b to purge any path to node j involving node k. Figure 3.1(c) also shows that when node i gets node k's update message, it updates its distance table through neighbor k, and checks for the possible paths to j through any other neighboring nodes; thus, it examines the available paths through its other neighbor b and updates the distance and the routing table entries accordingly. This results in the selection of the link (i, j) to the destination j (Figure 3.1(c)); again this is due to step(2), which purges all paths to destination j involving node k. When node

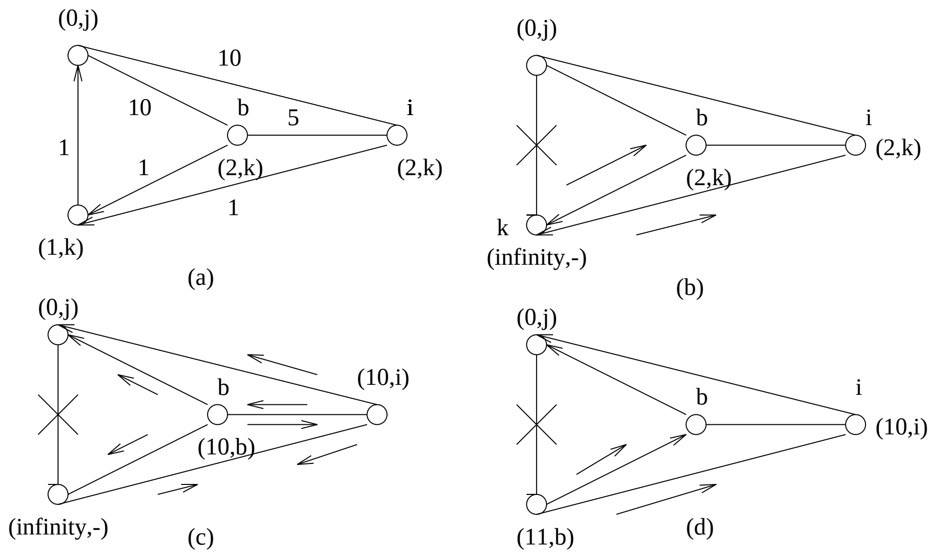

FIGURE 3.2: Example of PFA's operation

i receives node b's update reporting an infinite distance, node i need not have to update its routing table because it already has correct entries in its distance and routing tables (Figure 3.1(d)). Similarly, the updates that node k sends reporting a distance of 11 to node j will not impact on nodes i and b. This illustrates how step (2) (in the pseudocode) helps in the reduction of formation of temporary loops in explicit paths.

For PFA to work, some assumptions on the behavior of links and nodes must hold.

1. A lower-level protocol is responsible for maintaining the status of the link and handling of the transmission of messages.

2. The time interval during which a link is up is known as the *link up period* (LUP).

3. Messages are sent and received by a node only during a LUP.

4. There are no LUPs at a node when a node is down.

5. All nodes are initially down.

6. Messages received by a node are processed in the order of their arrival.

7. Link lengths are always positive and an infinite length represents a down link.

8. Time $T^{'}$ is defined such that between the time interval 0 and $T^{'}$ links and nodes go up and down and the cost of the link changes.

By assumption, a node processes any input event within a finite time; furthermore, each input event is processed independent of any other event. Therefore, there must be a time T when links have the same status at both ends and there are no changes after time T.

Definition 1 *The link weight d_{ij} for link(i,j) is extracted from the distance table D_v at a node v if there is a column k in D_v such that $d_{ij} = D^k_{vj} - D^k_{vi}$ and $p^k_{vj} = i$. Similarly, the link weight for link (i,j) can be extracted from the routing table if d_{ij} agrees $D^v_j - D^v_i$, where $p^v_i = i$.*

3.2 Correctness of PFA

In this section, we prove that PFA terminates in such a way that the distance to any other node maintained in the routing table in each node is the shortest distance of the final graph and the distance to any unreachable node is marked as undetermined.

Lemma 1 If a routing table is generated by PFA based on the distance table, any link weight that can be extracted from this routing table can be extracted from a column in the distance table.

Proof: Let d_{ij} be the link weight extracted from the routing table of node v. By Definition 1, the cost of a link can be extracted from the routing table as $(D^v_j - D^v_i)$, predecessor $p^v_j = i$ and successor $s^v_i = k$, from procedure *RT_Update*. Procedure *RT_Update* requires each distance in the routing table to be the minimum among the rows corresponding to the same destination in the distance table entry of a node as defined by the distributed Bellman-Ford algorithm. Therefore, the lemma is true.

Q.E.D.

Lemma 2 When a node comes up and initializes its distance table, the link weight that can be extracted from any of its distance table entries is the weight of the link.

Proof: A node coming up can be viewed as all the links connected to that node coming up. Initially, when a node is down, all its distance-table entries have an infinite distance. A link coming up is recorded as a single entry in the distance vector of the distance table, which is nothing but the weight of the link as given in Step (6) of the algorithm. Therefore, because the cost of any non-existent link can be assumed to be infinity, the link weight extracted from any column in the distance table of a node that comes up is the weight of that link. Therefore, the lemma is true.

Q.E.D.

Lemma 3 The change in the cost of a link will be reflected in the distance and the routing tables of a node adjacent to that link after a finite time T.

Proof: The change in the link cost can be due to a link coming up, a link going down, or the change in the link cost.

When a link comes up, a new column entry will be added to the distance table and the new link cost will be assigned to the corresponding entry in the distance table (Step (6)). Procedure RT_Update will be called, which eventually updates the routing table entry.

When a link goes down, the column entry will be deleted and the distance entries in the distance table will be set to ∞ (Step (5)). Procedure RT_Update again updates the routing table entries accordingly.

When the link cost changes, the distance entry in the distance table is updated to reflect the new link cost (Step (1) and Step (2)). These changes will be updated in the routing table again by procedure RT_Update.

From the assumptions, we know that all the link-cost changes occur in the time interval $[0, T')$. Therefore, there exists a time $T > T'$ when the adjacent node of a link updates its tables. This implies that link cost changes will be reflected in the distance and routing tables after a finite time T.

Q.E.D.

Property 1 After a finite time interval T, the routing table structures at all nodes will form the final shortest path.

Proof: The proof consists of the following two parts:

1. The old topology information present in node's routing and distance tables is updated.

2. The shortest-path trees are eventually computed.

Let the initial time be $T(0) = T$. Let $T(K)$ be the time by which all messages that are in transit at time $T(K-1), K \geq 0$, have arrived at their destination, and have been processed.

The proof is done by induction on K. At time $K = 0$, the property holds true. Assume that the property is true for $0 \leq K \leq M$.

A path of $M+1$ links is the concatenation of an adjacent link and a path with M links. From the assumptions, by time $T(M+1)$, the routing trees at time $T(M)$ of all nodes have been communicated to their neighboring nodes. By Lemma 3, these link cost changes will be updated in node's distance and routing tables within a finite time T. This proves the first part of the property.

The change in the link cost will result in a routing table update (in procedure RT_Update) as required. When a node has to select a new path, the minimum-in-row entry for that destination node is chosen from the distance table entries resulting in the shortest path in the final graph all along the way. This implies that the routing table structures at all nodes form the final shortest path.

Q.E.D.

Theorem 1 *If the distance entries in the distance and routing tables are finite, then a path can be extracted from the distance and routing table entries and this extracted path is loop-free.*

Proof: Let $T(K = 0)$ be the initial time when the algorithm begins execution. The theorem is true for $K = 0$ since no link exists between nodes at time $t = 0$.

Assume that the property is true for $T(M), 0 \leq M \leq K - 1$. By time $T(K)$, all the routing changes at time $T(K-1)$ would have been communicated to all nodes (assumption). No node will be marked as undetermined as all the distance entries are finite.

When a node comes up within the time $T(K - 1)$, step (6) of the algorithm will communicate this change in the link cost in finite time (by Lemma 3). As all the entries in the table are finite, a path can be extracted from any node i to any other node j by traversing through a node.

When a particular link is selected as a path from i to j, the loop-freedom of the path is checked in step 2 (procedure In_Path) and procedure RT_Update. An update message about link cost change will be sent to the neighbor. The loop-freeness of the update messages can be verified by traversing from destination node to the source node using predecessor information present in each entry of the distance and routing tables.

Therefore, the paths in the final graph are loop-free.

Q.E.D.

3.2.1 Distance Convergence

In this subsection, we prove that PFA terminates in such a way that the distance to any other node maintained in the routing table in each node is the shortest distance of the final graph and the distance to any unreachable node is marked as undetermined.

Property 2 If node j is not connected to node i in the final topology, then the distance between the two nodes is equal to infinity for all time after $T(H(i, \infty) + 1)$.

Proof: If a node i does not have a path to node j, the distance entries in node's tables will be set to ∞ (from the algorithm description). By definition, $H(i, d)$ gives the maximum number of links in the path from i whose distance to any other node is less than or equal to d in the final topology. This implies $H(i, d)$ is a finite quantity, because G is finite.

From Property 1, all the paths with links less than or equal to $H(i,\infty)$ will have their final length by time $T(H(i,\infty)+1)$. This proves Property 2.

Q.E.D.

Theorem 2 *The algorithm terminates in finite time after the last topological change happened.*

Proof: Assume that PFA does not terminate. This implies that there must be an infinite number of messages sent after the last topological change. These infinite messages must have finite distances since from Property 2 if the distance between the two nodes is equal to infinity, the algorithm converges. Moreover, from Theorem 1, the path extracted from the distance table must be a simple path. Thus, there must be some neighbor b that sends finite distances an infinite number of times to node i for node i to send messages without stopping.

Each time node i sends a message, it can be due to any one of the following reasons

1. It receives D^b_j from b and $D^i_j = D^b_j + d_{ib}$ where d_{ib} is the link weight
2. D^i_j has been in node i's distance table when it receives a message from b
3. neighbor b is in the path from i to j through another neighbor $k(\neq b)$ and $D^i_{jk} = D^i_{bk} + D^b_j$ (Step (2))

If the first case happens infinite times, node b sends D^b_j infinite times and $D^b_j = D^i_j - d_{ib} < D^i_j$ because $d_{ib} > 0$.

The second case can happen in a situation where D^i_j is not stable. This means that D^i_j is changed forever, which is similar to the first case in that there must be a neighbor b' such that b' sends $D^{b'}_j$ infinite times and $D^{b'}_j = D^i_j - d_{ib'} < D^i_j$, because $d_{ib'} > 0$. Else, if D^i_j becomes stable, then there must be an infinite number of times in which node i receives a distance that is shorter than D^i_j.

For the third case to happen an infinite number of times, D^i_{jk} must be changed forever. This in turn means that a neighbor has to send the distance vector D^b_j infinite times. This reduces to case (2) and eventually to case (1).

Consequently, there must be a neighbor b' sending $D_j^{b''}$ infinite times and $D_j^{b''} = D^i_{jb''} - d_{ib''} < D^i_j - d_{ib''} < D^i_j$ because $d_{ib''} > 0$.

Therefore, in all of the cases, there must be a node that will infinitely generate messages with a distance at least w less than D^i_j, where w is the minimum weight of the final graph. This will consequently contradict that all the distances are positive by recursively applying the above argument.

Q.E.D.

Property 3 When PFA terminates, all the link weights maintained in the distance table must be in the final graph.

Proof: This proof is done by induction. When a node comes up, its distance entries in the distance and routing tables are maintained correctly by Lemmas 1 and 2 and Property 1. If a link is not in the final graph, it implies that a node must have detected a link failure that caused it to delete the corresponding column entry from the distance table entry of the node and the distance is marked as infinity (step (5) of the algorithm). If the distance in the final graph d_{ij} is different from the earlier distance, node i must have been notified about this link-cost change by its neighbor. Thus, the correct distance entries are maintained in the final graph for all adjacent nodes.

Assume that the result is true for nodes that are k hops away from i.

We will show by induction that the result is true for nodes that are $k+1$ hops away from i. Let j be a node that is $k+1$ hops away from node i and a be a node that is k hops away from i. Since all nodes that are k hops away from i maintain the distance entries correctly, the distance entry is correct for node a. The distance from j to i is the sum of d_{ja} and D^a_i (step 1 of the algorithm). This is nothing but the minimum in row of the distances from i to j and hence is the shortest-path from i to j. Therefore, this distance entry will be present in the final graph unless the link has gone down before the algorithm terminates in which case, an infinite distance will be maintained. This proves the property.

Q.E.D.

Theorem 3 *When PFA terminates, the distance for any node i to any other node j in the routing table of node i is the shortest distance from i to j in the final graph and the successor will be maintained correctly; furthermore, the distance from node i to any unreachable node is marked as undetermined.*

Proof: We prove the theorem by induction.

From Lemma 3, the weight of any link must be maintained by its adjacent node. When a link comes up, the cost of the link will be assigned to the distance table entry (neighbor node) and the predecessor will be initialized to be the source node itself (step 6 of the algorithm). The distance table is checked to see whether its distance entry is smaller than the routing table entry and the routing table is updated according to procedure RT_Update with successor and predecessor entries properly set. If a link is in the path to the destination through any other neighboring nodes, then the distance and the routing table entries are also updated.

Assume that the result is true for any node j which is k hops away from node i. We will show by induction that the result is true for a node $k+1$ hops away from i. Consider any node j that is $k+1$ hops away from i. There must be a neighbor b of node j that is k hops away from i and that maintains correct distance and routing table entries. Let d_{jb} denote the distance between node j and its neighbor node b which are $k+1$ and k hops away from i respectively. Let D^i_b be the distance from i to b. D^i_b is the shortest path from i to b as D^i_b is the minimum in row of b and each distance table entry represents an existent path.

Since b is a neighbor of j, $D^i_j = D^i_b + d_{bj}$ is the shortest path from i to j, with d_{bj} being the minimum in row entry. The predecessor path will also be maintained correctly (from step 1). Furthermore, any node x in the shortest path from i to j must also have the subpath from x to i as the shortest path because it is the minimum in the row of x.

Procedure RT_Update is called in the update routine after updating all the distance table entries of that node. This routine picks up a minimum entry through one of its neighbors and will have a successful trace for the destination node j and thus will have $D^i_j = D^i_{ji'} = D^{i'}_j + d_{ii'} = D^i_j$ and $s^i_j = b$.

Q.E.D.

3.3 Complexity of PFA

The number of messages generated is bounded by an exponential function on V, the number of nodes, a polynomial of degree E, and a linear function of the number of topological changes. Some of the advantages/improvements of PFA are

- The storage required is the same as that of previous path-finding algorithms for each node.
- PFA can detect network partitions faster than any previous path-finding algorithm, because a node updates its entire distance table for each update message received from its neighbor.
- Counting-to-infinity problem is eliminated.
- PFA's time complexity is O(h) in the worst-case where, h is the height of the routing tree.

Theorem 4 below proves this result.

Time complexity is defined as the largest time that can elapse between the moment T when the last topology change occurs and the moment at which all nodes have final shortest path and distances to all other nodes. *Communication complexity* is defined as the maximum number of node identities exchanged (messages) after time T before the final graph is reached.

Consider Figure 3.3. The weight of the links are as indicated. Assume n_d is the destination node. Node n_1, n_2, n_3 and n_4 will have the shortest path node n_x before link (n_d,n_x) fails. After the link failure, nodes n_1, n_2, n_3 and n_4 immediately identify that the only possible way to reach the destination node n_d is through the link (n_i,n_d) for i = 1,2,3,4 upon receiving an update message from node n_x about the link failure, instead of going through an intermediate step of selecting the path through nodes n_2, n_3 and n_4 respectively as in

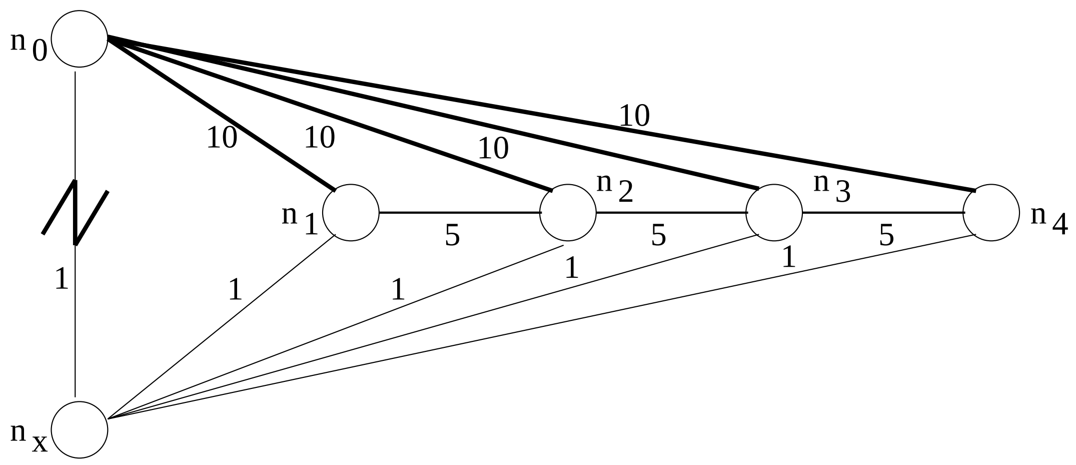

FIGURE 3.3: Complexity of PFA

the case of any other path-finding algorithm. That is, a node need not have to wait for an update message from the neighbor n_2, n_3 and n_4 before arriving at the final graph. This reduces the number of update messages required.

Theorem 4 *The time complexity for a single failure or change for PFA is O(h) in the worst-case, where h is the maximum height of the routing tree experienced during the computation.*

Proof: Let the source node be i and the destination node be j. Let the failed link be (n, m) and node m is downstream node to node n. There are four possible situations involving the shortest path.

1. (n, m) is not on the shortest path and its length does not change enough to change the shortest path.

2. (n, m) is not on the shortest path and its length decreases enough that it becomes part of the shortest path.

3. (n, m) is on the shortest path and its length does not change enough to modify the shortest path (although the length of the shortest path changes).

4. (n, m) is on the shortest path and its length increases enough that the shortest path changes.

A node with the initial shortest path not going through the changed link (Case (1)) does not change its routing table since the original shortest path is not changed and the change in link cost has resulted only in the increase in the path length through other routes.

In Case (2), a node will be aware of the change in link cost along the shortest path after a delay not exceeding the number of links on the new shortest path. In Case (3) the change will be noticed in the worst case after a delay of at most the number of links in the shortest path.

Let node n_k with the original shortest path through the changed link be k hops away from node n on the initial shortest path. When a link cost changes or the link fails, node containing the failed link selects a new neighbor (changes the successor) for a path to destination j. This changes the routing table entry at node n and the routing vector generated due to link failure will be sent to all its neighbors. Each of these neighbors will update their table entries and the change in link cost propagates. This process continues until a stable node which does not change its successor is encountered. The tables are updated either on the receipt of the update message or if the distance update message received from a node's neighbor has any effect on node's other distance table entries. The distance of the stable node found in the path from i to j in the new shortest path is bounded by h, the height of the tree. Therefore, in the worst case, the number of steps required for a node to converge to its correct distance is O(h).

Q.E.D.

3.4 Comparison with Humblet's Algorithm

Humblet [Hum91] has presented a path-finding algorithm in which a breadth-first-search on the nodal distance tables is done to construct the routing tables. He has also pointed out how other previous path findings algorithms [RF91, CRKGLA89] compare with his.

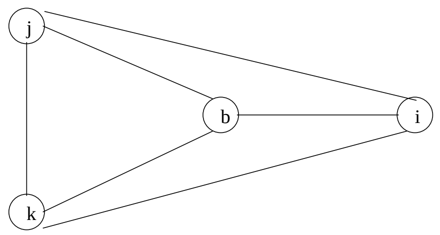

FIGURE 3.4: Example Topology

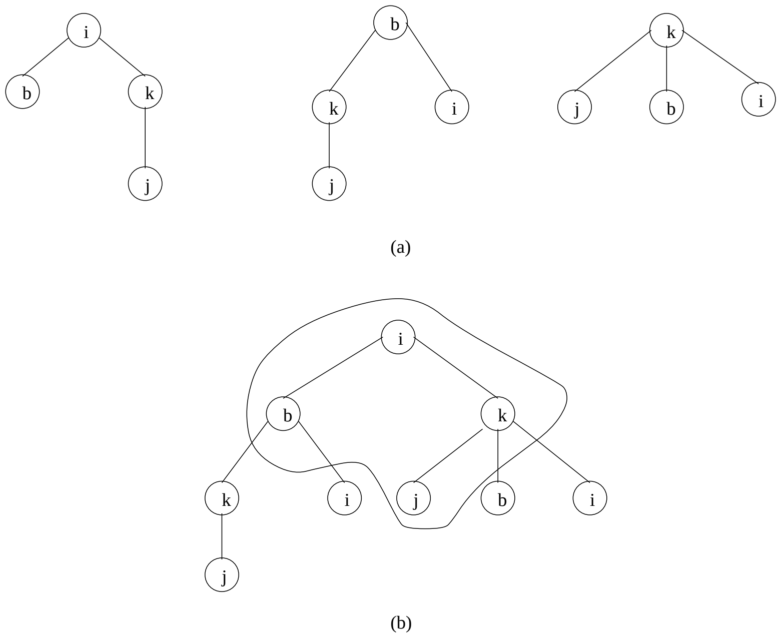

FIGURE 3.5: Routing table at node i for Humblet's Algorithm

The comparison of the algorithms are done based on cost of the links. We illustrate the differences between this algorithm and PFA using Figure 3.4, 3.5, 3.6.

As explained earlier, the predecessor information in node's distance table column entry and the routing table entries can be used to derive the complete path to any destination. The union of all the derived paths in any column of the distance table or routing table at any node forms a tree (Theorem 1). The individual node routing trees obtained from Humblet's algorithm for the topology of Figure 3.4 is shown in Figure 3.5(a). The algorithm first attaches the trees in Figure 3.5(a), derived from each distance table column to node i itself and then performs a breadth-first-search on the combined tree. Also, each node can be visited only once. The resulting configuration is as in Figure 3.5(b).

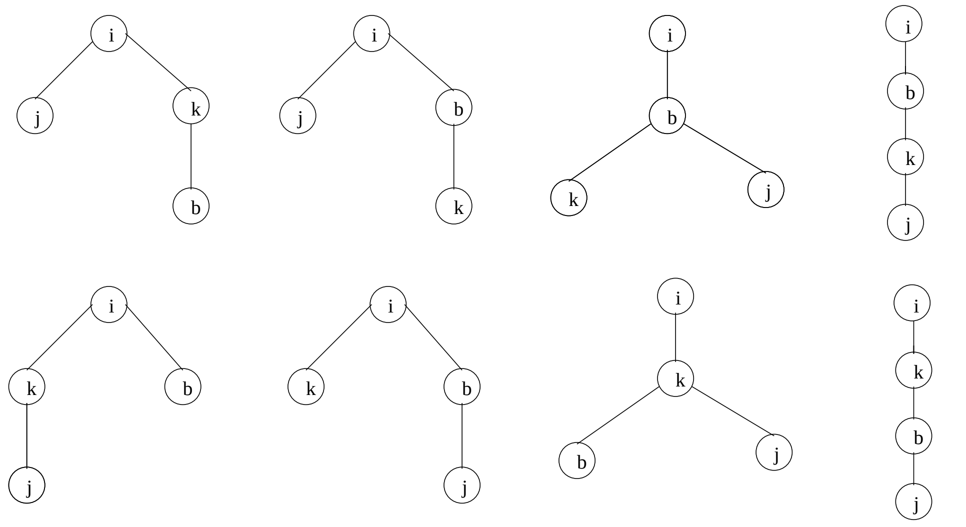

FIGURE 3.6: Routing table at node i for the Path-Finding Algorithm

According to PFA, the possibilities of the resulting routing table at node i are given in Figure 3.6. It can be seen that in Humblet's algorithm, at any node, the subtree with the root other than node itself in the routing table tree reside in the single column in the distance table. In contrast, PFA does not have any such restrictions during the period of time before the algorithm converges. This gives additional flexibility in the selection of the routes. Also, if there are two parameters that are required to be minimized, say distance and the number of hops, then a path can be selected from the routing table according to the given specifications. Our algorithm converges faster than Humblet's algorithm, because the routing-table tree can be the final shortest-path tree for node i to all the other nodes in the given topology.

3.5 Summary

In this chapter, we have presented a new path-finding algorithm that reduces the occurrence of temporary routing loops without the need for internodal synchronization mechanisms. The correctness, convergence and the complexity of the algorithm was presented in detail. The performance of PFA is compared with Humblet's path-finding algorithm qualitatively. Compared to Humblet's algorithm, PFA has the advantage of having fewer temporary loops and has a worst-case complexity of O(h) for a single resource recovery or failure, h being the height of the routing tree.

Chapter 4
Loop-Free Path-Finding Algorithm

In this chapter, we present a path-finding algorithm (LPA) that is loop-free at every instant. This is the first algorithm that eliminates the formation of temporary loops without the need for inter-nodal synchronization spanning over multiple hops or the specification of complete path information.

LPA is built on two basic mechanisms – using predecessor information to eliminate counting-to-infinity and blocking temporary routing loops using an inter-neighbor synchronization method similar to the one proposed in [GLA92].

Using the predecessor information, each router can infer if the path corresponding to a distance-table or routing-table entry includes the router itself, i.e., if there is a loop in the path offered by a neighbor. This feature eliminates the counting-to-infinity problem present in DBF. Furthermore, a router detects a temporary loop within a finite time that depends on the speed with which correct predecessor information reaches a router, and not on the distance values of the paths offered by its neighbors; therefore, temporary loops are detected much faster than in DBF and its variations.

In LPA, a router which decides that a loop may be formed if it changes its successor is asked to block such a loop by reporting to all its neighbors an infinite distance for a destination, and by waiting for those neighbors to acknowledge its message with their own distances and predecessor information, before a router changes its successor. Because of the overhead involved, a router should not send a query every time it has to change its successor to a destination; a router decides when to block a potential loop by comparing the distances reported by its neighbors against a *feasible distance*, defined to be the smallest value achieved by a router's own distance since the last query sent by a router. A router is forced to block a potential loop with a query only when no neighbor reports a distance smaller than router's own feasible distance; this feature accounts for the low overhead incurred in LPA to accomplish loop-free paths at every instant.

In contrast to many prior loop-free routing algorithms [GLA92, GLA93, JM82], queries propagate only one hop in LPA. Furthermore, updates and routing table entries in LPA require a single node identifier as path information rather than a variable number of node identifiers as in prior algorithms [GLA92].

The rest of the chapter is organized as follows. The next section gives a description of LPA and an example illustrating key aspects of LPA's operation; Sections 4.3 and 4.4 provide a detailed proof of LPA's loop-freedom and convergence to correct routing table entries, respectively; Section 4.5 addresses the complexity of LPA; and finally, Section 4.6 summarizes the results.

4.1 LPA Description

Each router maintains a *distance table*, a *routing table* and a *link-cost table*. The distance table at each router i is a matrix containing, for each destination j and for each neighbor k of router i, the distance and the predecessor reported by router k, denoted by D^i_{jk} and p^i_{jk}, respectively.

The routing table at router i is a column vector containing, for each destination j the minimum distance (denoted by D^i_j), predecessor (denoted by p^i_j), successor (denoted by s^i_j), and a marker (denoted by tag^i_j) used to update the routing table. For destination j, tag^i_j specifies whether the entry corresponds to a simple path ($tag^i_j = correct$), a loop ($tag^i_j = error$) or a destination that has not been marked ($tag^i_j = null$).

The link-cost table lists the cost of each link adjacent to a router. The cost of the link from i to k is denoted by d_{ik} and is considered to be infinity when a link fails.

An update message from router i consists of a vector of entries; each entry specifies an update flag (denoted by u^i_j), a destination j, the reported distance to that destination (denoted by RD^i_j), the reported predecessor in the path to the destination (denoted by rp^i_j). The update flag indicates whether the entry is an update ($u^i_j = 0$), a query ($u^i_j = 1$) or a reply to a query ($u^i_j = 2$). The distance in a query is always set to ∞.

The implicit path information from a router to any destination can be extracted from the predecessor entries of router's distance and routing tables. In the specification of LPA, the successor to destination j for any router is simply referred to as the successor of a router, and the same reference applies to other information maintained by a router. Similarly, updates, queries and replies refer to destination j, unless stated otherwise.

Figures 4.1 and 4.2 specify LPA in pseudocode. The rest of this section provides an informal description of LPA.

The procedures used for initialization are *Init1* and *Init2*; Procedure *Message* is executed when a router processes an update message; procedures *linkUp*, *linkDown* and *linkChange* are executed when a router detects a new link, the failure of a link, or the change in the cost of a link. We refer to these procedures as event-handling procedures. For each entry in an update message, Procedure *Message* calls procedure *Update*, *Query*, or *Reply* to handle an update, a query, or a reply, respectively. An important characteristic of all the event-handling procedures is that they mark $tag^i_j = null$ for each destination j affected by the input event.

When router i receives an input event regarding neighbor k (an update message from neighbor k or a change in the cost or status of link (i,k)) it updates its link-cost table with the new value of link d_{ik} if needed, and then executes procedure DT. This procedure updates $D^i_{jk} = D^k_j + d_{ik}$ and $p^i_{jk} = p^i_k$ for each destination j affected by the input event. In addition, it determines whether the path to any destination j through any of other neighbor of router i includes neighbor k. If the path implied by the predecessor information reported by router b to destination j includes router k, then the distance entry of that path is updated as $D^i_{jb} = D^i_{kb} + D^k_j$ and the predecessor entry is updated as $p^i_{jb} = p^k_j$.

After procedure DT is executed, the way in which router i continues to update its routing table for a given destination depends on whether it is *passive* or *active* for that destination. A router is passive if it has a *feasible successor*, or has determined that no such successor exists and is active if it is searching for a feasible successor. A feasible successor for router i with respect to destination j is a neighbor router that satisfies the feasibility condition (FC). When router i is passive, it reports the current value of D^i_j in all its updates and replies. However, while router i is active, it sends an infinite distance in its replies and queries. An active router cannot send an update regarding the destination for which it is active, this is because an update during active state would have to report an infinite distance to ensure that the inter-neighbor synchronization mechanism used in LPA provides loop freedom at every instant.

Feasibility Condition (FC): If at time t, router i needs to update its current successor, it can choose as its new successor $s^i_j(t)$ any router $n \in N_i(t)$ such that i $D^i_{jn}(t) + d_{in}(t) = D_{min}(t) = Min\{D^i_{jx}(t) + d_{ix}(t) | x \in N_i(t)\}$ and $D^i_{jn}(t) < FD^i_j(t)$. If no such neighbor exists and $D^i_j(t) < \infty$, router i must keep its current successor. If $D_{min}(t) = \infty$ then $s^i_j(t) = null$.

The successor graph for destination $j \in G$, denoted by $S_j(G)$, is a directed graph in which nodes are the same nodes of G and where directed links are determined by the successor entries in the nodal routing tables. Loop freedom is guaranteed at all times in G if $S_j(G)$ is always a directed acyclic graph. If G is connected in steady state, when all routing tables are correct, $S_j(G)$ must be a directed tree whose links point to j.

If router i is passive when it processes an update for destination j, it determines whether or not it has a feasible successor, i.e., a neighbor router that satisfies FC.

If router i finds a feasible successor, it sets FD^i_j equal to the smaller of the updated value of D^i_j and the present value of FD^i_j. In addition, it updates its distance, predecessor, and successor using procedure TRT. This procedure ensures that any finite distance in the routing table corresponds to a simple path by selecting as the successor to any destination j a neighbor k that satisfies the following property:

Property 4 Router i sets $s^i_j = k$ at time t only if $D^i_{xk}(t) \leq D^i_{xp}(t)$ for every neighbor p other than k and for every node x in the path from i to j defined by the predecessors reported by neighbor k.

Let $P^i_{jk}(t)$ denote the path from k to j defined by the predecessors reported by neighbor k to router i and stored in router i's distance table at time t. Procedure *TRT* implements Property 4 by traversing $P^i_{jk}(t)$ from j back to k using the predecessor information. This path traversal ends when either a predecessor x is reached for which tag^i_x = *correct* or *error*, or neighbor k is reached. If tag^i_x = *error*, then tag^i_j is set to *error* too; otherwise, the neighbor k or a correct tag must be reached, in which case tag^i_j is set to *correct*.

After updating its routing table, router i prepares an update to its neighbors if its routing table entry changes.

Alternatively, if router i finds no feasible successor, then it updates $FD^i_j = \infty$ and updates its distance and predecessor to reflect the information reported by its current successor. If $D^i_j(t) = \infty$, then $s^i_j(t)$ = null. Router i also sets the reply status flag ($r^i_{jk} = 1$) for all $k \in N_i$ and sends a query to all its neighbors. Router i is then said to be *active*, and cannot change its path information until it receives all the replies to its query.

Queries and replies are processed in a manner similar to the processing of an update described above. If the input event that causes router i to become active is a query from its neighbor k, router i sends a reply to router k reporting an infinite distance. This is the case, because router k's query, by definition, reports the latest information from router k, and router i will send an update to router k when it becomes passive if its distance is smaller than infinity. A link-cost change is treated as a number of updates.

Once router i is active for destination j, it may not have to do anything more regarding that destination after executing procedures *RT* and *DT* as a result of an input event. However, when router i is active and receives a reply from router k, it updates its distance table and resets the reply flag ($r^i_{jk} = 0$).

Router i becomes passive at time t when $r^i_{jk}(t) = 0$ for every $k \in N_i$. At that time, router i can be certain that all its neighbors have processed its query reporting an infinite distance and router i is therefore free to choose any neighbor that provides the shortest distance, if there is any; or router i has found a feasible successor through one of its neighbors $k \in N_i$. If such a neighbor is found, router i updates the routing table as the minimum in distance-table row for destination j and also updates $FD^i_j = D^i_j$.

A router does not wait indefinitely for replies from its neighbors because a router replies to all its queries regardless of its state. Thus, there is no possibility of deadlocks due to the inter-neighbor coordination mechanism.

If router i is passive and has already set $D^i_j = \infty$ and receives an input event that implies an infinite distance to j, then router i simply updates D^i_{jk} and d_{ik} and sends a reply to router k with an infinite distance if the input event is a query from router k. This ensures that updates messages will stop in G when a destination becomes unreachable.

Router i initializes itself in passive state with an infinite distance for all its known neighbors and with a zero distance to itself. After its initialization, router i sends updates containing the distance to itself to all its neighbors.

When router i establishes a link with a neighbor k, it updates its link-costs table and assumes that router k has reported infinite distances to all destinations and has replied to any query for which router i is active; furthermore, if router k is a previously unknown destination, router i initializes the path information of router k and sends an update to the new neighbor k for each destination for which it has a finite distance. When router i is passive and detects that link (i, k) has failed, it sets $d_{ik} = \infty$, $D^i_{jk} = \infty$ and $p^i_{jk} = \text{null}$; after that, router i carries out the same steps used for the reception of a link-cost change message in passive state. When router i is active and loses connectivity with a neighbor k, it resets the reply flag and resets the path information i.e., assumes that the neighbor k sent a reply reporting an infinite distance.

It follows from this description of router i's operation that the order in which router i processes updates, queries and replies does not change with the establishment of new links or link failures. The addition or failure of a router is handled by its neighbors as if all the links connecting to that router were coming up or going down at the same time.

```
Procedure Init1
when router $i$ initializes itself
do begin
    set a link-state table with costs of adjacent links;
    $N \leftarrow \{i\}; N_i \leftarrow \{x \mid d_{ix} < \infty\}$;
    for each $(x \in N_i)$
    do begin
        $N \leftarrow N \cup x$; $tag_x \leftarrow$ null;
        $s^i_x \leftarrow$ null; $p^i_x \leftarrow$ null;
        $D^i_x \leftarrow \infty$; $FD^i_x \leftarrow \infty$
    end
    $s^i_i \leftarrow i$; $p^i_i \leftarrow i$; $tag^i_i \leftarrow$ correct;
    $D^i_i \leftarrow 0$; $FD^i_i \leftarrow 0$;
    for each $j \in N$ call Init2$(x, j)$;
    for each $(n \in N_i)$ do add $(0, i, 0, i)$ to $LIST_i(n)$;
    call Send
end


Procedure Init2$(x, j)$
begin
    $D^i_{jx} \leftarrow \infty$; $p^i_{jx} \leftarrow null$; $s^i_{jx} \leftarrow null$; $r^i_{jx} \leftarrow 0$
end


Procedure Send
begin
        for each $(n \in N_i)$
        do begin
            if $(LIST_i(n)$ is not empty$)$
            then send message with $LIST_i(n)$ to $n$
            empty $LIST_i(n)$
        end
end


Procedure Message
when router $i$ receives a message on link $(i, k)$
begin
        for each entry $(u^k_j, j, RD^k_j, rp^k_j)$ such that $j \neq i$
        do begin
            if $(j \notin N)$
            then begin
                    if $(RD^k_j = \infty)$ then delete entry
                      else begin
                            $N \leftarrow N \cup \{j\}$; $FD^i_j = \infty$;
                            for each $x \in N_i$ call Init2$(x, j)$
                            $tag^i_j \leftarrow null$; call DT$(j, k)$
                      end
            end
            else begin
                    $tag^i_j \leftarrow null$; call DT$(j, k)$
            end
        end
        for each entry $(u^k_j, j, RD^k_j, rp^k_j)$ left
        such that $j \neq i$
        do case of value of $u^i_j$
            0:  [Entry is an update]
                call Update$(j, k)$
            1:  [Entry is a query]
                call Query$(j, k)$
            2:  [Entry is a reply]
                call Reply$(j, k)$
        end
        call Send
end


Procedure Link_Up $(i, k, d_{ik})$
when link $(i, k)$ comes up do begin
        $d_{ik} \leftarrow$ cost of new link;
        if $k \notin N$ then begin
                        $N \leftarrow N \cup \{k\}$; $tag^i_k \leftarrow$ null;
                        $D^i_k \leftarrow \infty$; $FD^i_k \leftarrow \infty$;
                        $p^i_k \leftarrow$ null; $s^i_k \leftarrow$ null;
                        for each $x \in N_i$ do call Init2$(x, k)$
        end
        $N_i \leftarrow N_i \cup \{k\}$;
        for each $j \in N$ do call Init2$(k, j)$;
        for each $j \in N - k \mid D^i_j < \infty$ do add $(0, j, D^i_j, p^i_j)$ to $LIST_i(k)$;
        call Send
end


Procedure Link_Down$(i, k)$
when link $(i, k)$ fails do begin
        $d_{ik} \leftarrow \infty$;
        for each $j \in N$ do begin
            call DT$(j, k)$;
            if $(k = s^i_j)$ then $tag^i_j \leftarrow null$
        end
        delete column for $k$ in distance table; $N_i \leftarrow N_i - \{k\}$;
        delete $r^i_{jk}$;
        for each $j \in (N - i) \mid k = s^i_j$ do begin
            call Update$(j, k)$
        end
        call Send
end


Procedure Link_Change $(i, k, d_{ik})$
when $d_{ik}$ changes value do begin
        $old \leftarrow d_{ik}$;
        $d_{ik} \leftarrow$ new link cost;
        for each $j \in N$ do begin
            call DT$(j, k)$;
            for each $j \in N$
            do if $(D^i_j > D^i_{jk}$ or $k = s^i_j)$ then $tag^i_j \leftarrow$ null
        end
        for each $j \in N$ do begin
            if $(d_{ik} < old)$
            then for each $j \in N - i \mid D^i_j > D^i_{jk}$ do call Update$(j, k)$;
            else for each $j \in N - i \mid k = s^i_j$ do call Update$(j, k)$
        end
        call Send
end


Procedure DT$(j, k)$
begin
        $D^i_{jk} \leftarrow RD^k_j + d_{ik}$; $p^i_{jk} \leftarrow rp^k_j$;
        for each neighbor $b$ do begin
            $h \leftarrow j$;
            while $(h \neq i$ or $k$ or $b)$ do $h \leftarrow p^b_h$;
            if $(h = k)$ then begin
                $D^i_{jb} \leftarrow D^i_{kb} + RD^k_j$; $p^i_{jb} \leftarrow rp^k_j$
            end
            if $(h = i)$ then begin
                $D^i_{jb} \leftarrow \infty$; $p^i_{jb} \leftarrow null$
            end
        end
end
```

FIGURE 4.1: LPA Specification

4.2 Example

As an example of LPA's operation and its loop-freedom property, consider the five-node network depicted in Figure 4.3. In this network, links and nodes have the same processing

```
Procedure Update($j$, $k$)
begin
        if ($r^i_{jx} = 0, \forall x \in N_i$)
        then begin
                if (($s^i_j = k$) or ($D^i_{jk} < D^i_j$))
                then call PU($j$)
        end
        else call AU($j$, $k$)
end

Procedure Reply($j$, $k$)
begin
        $r^i_{jk} \leftarrow 0$;
        if ($r^i_{jn} = 0, \forall n \in N_i$)
        then if (($\exists x \in N_i \mid D^i_{jx} < \infty$) or ($D^i_j < \infty$))
                then call PU($j$)
        else call AU($j$, $k$)
end

Procedure Query($j$, $k$)
begin
        if ($r^i_{jx} = 0 \forall x \in N_i$)
        then begin
                if ($D^i_j = \infty$ and $D^i_{jk} = \infty$)
                then add (2, $j$, $D^i_j$, $p^i_j$) to $LIST_i(k)$
                else begin
                        call PU($j$);
                        add (2, $j$, $D^i_j$, $p^i_j$) to $LIST_i(k)$;
                end
        else call AU($j$, $k$)
end

Procedure AU($j$, $k$)
begin
        if ($k = s^i_j$) then begin
                $D^i_j \leftarrow D^i_{jk}$; $p^i_j \leftarrow p^i_{jk}$
        end
end
```

```
Procedure PU($j$)
begin
        $DT_{min} \leftarrow Min\{D^i_{jx}\ \forall\ x \in N_i\}$;
        $FCSET \leftarrow \{n \mid n \in N_i,\ D^i_{jn} = DT_{min},\ D^n_j < FD^i_j\}$;
        if ($FCSET \neq \emptyset$) then begin
          call TRT($j$, $DT_{min}$); $FD^i_j \leftarrow Min\{D^i_j, FD^i_j\}$
        end
        else begin
                $FD^i_j = \infty$; $r^i_{jx} = 1\ \ \forall x \in N_i$; $D^i_j = D^i_{j\, s^i_j}$; $p^i_j = p^i_{j\, s^i_j}$;
                if ($D^i_j = \infty$) then $s^i_j \leftarrow$ null;
                $\forall$ x $\in N_i$
                do begin
                        if (query and x = k)
                        then $r^i_{jk} \leftarrow 0$
                        else add (1, $j$, $\infty$, null) to $LIST_i(x)$
                end
        end
end

Procedure TRT($j$, $DT_{min}$)
begin
        if ($D^i_{j\, s^i_j} = DT_{min}$)
        then $ns \leftarrow s^i_j$
        else $ns \leftarrow b \mid \{b \in N_i$ and $D^i_{jb} = DT_{min}\}$;
        $x \leftarrow j$;
        while ($D^i_{x\ ns} = Min\{D^i_{xb}\ \ \forall\ \ b \in N_i\}$ and $D^i_{xns} < \infty$ and $tag^i_x = null$)
        do $x \leftarrow p^i_{x\ ns}$;
        if ($p^i_{x\ ns} = i$ or $tag^i_x$ = correct)
        then $tag^i_j \leftarrow$ correct  else $tag^i_j \leftarrow$ error
        if ($tag^i_j$ = correct)
        then begin
                if ($D^i_j \neq DT_{min}$ or $p^i_j \neq p^i_{j\ ns}$) then
                add (0, $j$, $DT_{min}$, $p^i_{j\ ns}$) to $LIST_i(x)$  $\forall x \in N_i$;
                $D^i_j \leftarrow DT_{min}$; $p^i_j \leftarrow p^i_{j\ ns}$; $s^i_j \leftarrow ns$
        end
        else begin
                if ($D^i_j < \infty$)
                then add (0, $j$, $\infty$, null) to $LIST_i(x)$  $\forall x \in N_i$;
                $D^i_j \leftarrow \infty$; $p^i_j \leftarrow$ null; $s^i_j \leftarrow$ null
        end
end
```

FIGURE 4.2: LPA Specification (Continued)

or propagation delays; Q represents the queries, R replies and U indicates updates. The operation of the algorithm is discussed for the case in which the cost of ink (a, j) changes. The arrowhead from node x to node y indicates the that node y is the successor of node x towards the destination j (i.e., $s^x_j = y$). The label in parenthesis assigned to node x indicates the feasible distance from x to j (FD^x_j), current distance (D^x_j), and predecessor of the path from x to j (p^x_j). Steps 1 through 5 of figure 4.3 depicts the behavior of LPA. Updates and replies are followed by the value of RD^x_j and rp^x_j in parentheses. Nodes in the

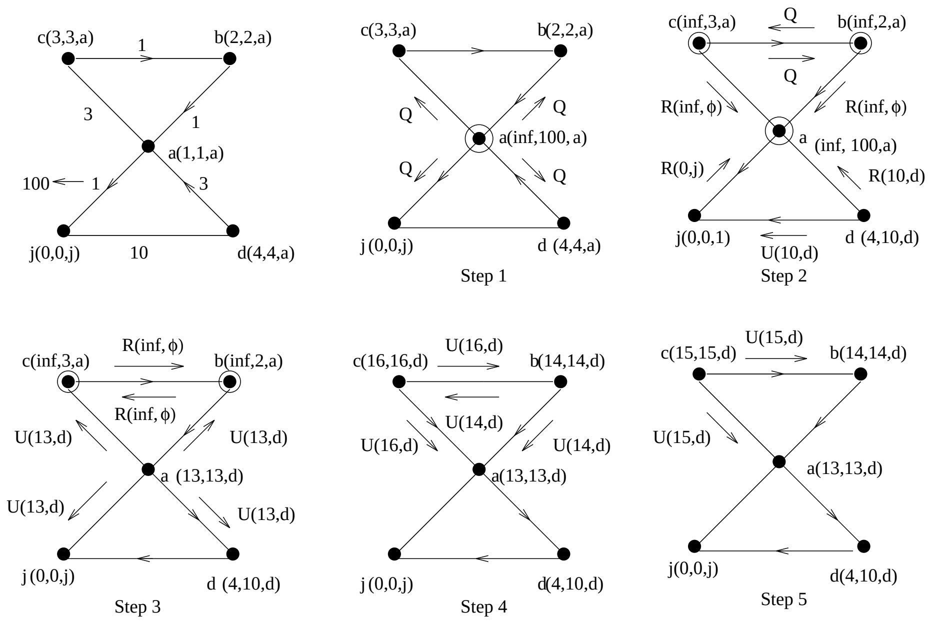

FIGURE 4.3: Example of LPA's Operation

active state are indicated with a circle around them. FD_j^i is always decreasing as long as node i is in the active state.

When node a detects the change in the cost of link (a, j), it determines that it does not have a feasible successor as none of its neighbors have a distance smaller than $FD_j^a = 1$. Accordingly, node a becomes active and sends a query to all its neighbors (Step 1 in Figure 4.3).

Nodes b and c also recognize that they do not have a feasible successor. This is achieved in a single step as the node traces through all its neighbors on receipt of an input event. Node b (c) becomes active and sends query to c (b) and reply to a. On the other hand, node d is able to find a path to j and replies with the cost of the alternate path to j to node a's query and updates its distance to j maintaining the same feasible distance.

When node a receives replies from all its neighbors, it becomes passive again, and replies to the queries of nodes b and c with its feasible distance. Having found their feasible successor, nodes b and c update their path information accordingly. All nodes exchange update messages informing the new path information with their neighbors (Step 4) and the final stable topology is shown in Step 5.

4.3 Loop Freedom in LPA

It is clear that $S_j(G)$ would be loop free at every instant if a router sent a query reporting an infinite distance to its neighbors every time it needed to change successors, because no router would change it before blocking any potential loop by sending an infinite distance "upstream" the loop. However, it is not obvious that loop freedom is maintained at every instant when routers use the feasibility condition FC to decide if they have to send a query before changing $S_j(G)$. The following theorem shows that this is the case, i.e., that LPA is free of loops at every instant. The proof is by contradiction and is essentially the same as the one presented in [GLA92] for another algorithm.

Proposition 1: *If a loop is formed in the successor graph $S_j(G)$ for the first time at time t, then some router i in that loop must choose an upstream router as its successor at time t.*

By assumption, $S_j(G)$ is a directed acyclic graph before the loop is formed at time t. If a loop has to be formed at time t, there must be at least one router $k \in S_j(G)$ that changes its successor because the successor information can be changed only when an update occurs or when a router detects a change in a link cost or status. This implies that an upstream router will be chosen by some router x in the loop.

Q.E.D.

Theorem 5 *In a network G, the successor graph $S_j(G)$ is loop-free at every instant t.*

Proof: The proof is by contradiction to the feasibility condition FC.

Let G be a stable topology and let the successor graph $S_j(G)$ be loop-free at every instant before t. Let $C_j(t)$ be the loop formed in the successor graph at time t. It is evident that no loops can be created unless routers change successors and modify the successor

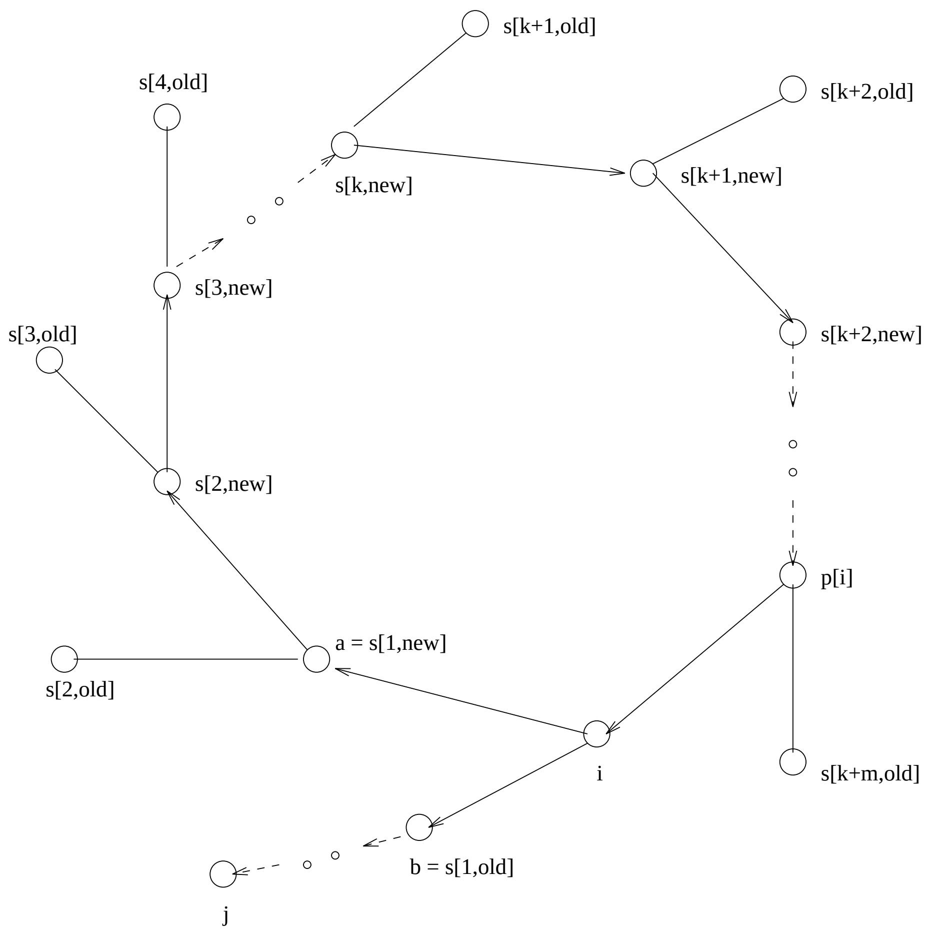

FIGURE 4.4: Loop in G

graph $S_j(G)$, and it follows from proposition 1 that at least one router must change its successor at time t and choose an upstream neighbor for a loop to be formed.

At time $t = 0$, when the network is first initialized, each router knows only how to reach itself. This is equivalent to saying that at time 0, $S_j(G)$ is a disconnected graph of one or more components, each with a single router. Therefore $S_j(G)$ is loop-free at time $t = 0$.

Let $t > 0$, and assume that a loop $C_j(t)$ is formed when router i makes router a ($=s[1, new]$) its new successor (Figure 4.4). This implies the path from a to j at time t, denoted by $P_{aj}(t)$, includes $P_{ai}(t)$.

Let path $P_{ai}(t)$ consist of a chain of routers $\{a, s[2, new], ..., i\}$, as shown in the Figure 4.4. Router $s[k, new]$ is the kth hop in the path P_{ai} at time t and $s[k + 1, new]$ is its successor at time t. Router $s[k, new]$ sets $s_j^{s[k,new]} = s[k+1, new]$ at time $t_{s[k+1,new]} \leq t$ and makes no more updates to its successor in the time interval $(t_{s[k+1,new]}, t]$; therefore, $s_j^{s[k,new]}(t_{s[k+1,new]}) = s_j^{s[k,new]}(t)$ and $D_j^{s[k,new]}(t_{s[k+1,new]}) = D_j^{s[k,new]}(t)$.

Similarly, router $s[k+1, old]$ is router $s[k, new]$'s successor just before node $s[k, new]$ becomes the kth hop of path $P_{ai}(t)$ by making router $s[k+1, new]$ its successor at time $t_{s[k+1,new]} \leq t$.

Because all routers in $C_j(t)$ must have a successor at time t, all of them must be passive at that time. If all routers in $C_j(t)$ have always been passive before time t, it follows from Theorem 1 in [GLA92] that router i cannot create $C_j(t)$; the proof of that theorem is based on the fact that FD_j^i can only decrease as long as router i is passive. The rest of the proof needs to show that $C_j(t)$ cannot be formed if at least one router in $P_{ai}(t)$ was temporarily active before time t.

Consider the case in which node $s[k, new] \in P_{aj}(t)$ is already passive before it updates its distance and successor to join $P_{aj}(t)$ at time $t_{s[k+1,\ new]} \leq t$. According to LPA, $D_j^{s[k,\ new]}(t_{s[k+1,\ new]}) = RD_j^{s[k,\ new]}(t_{s[k+1,\ new]})$; furthermore, according to FC it must be true that

$$\begin{aligned} D_{j\ s[k+1,\ new]}^{s[k,\ new]}(t_{s[k+1,\ new]}) \quad &= \quad D_{j\ s[k+1,\ new]}^{s[k,\ new]}(t) < FD_j^{s[k,\ new]}(t) \\ &\leq D_j^{s[k,\ new]}(t_{s[k+1,\ old]}). \end{aligned}$$

Hence, if router $s[k-1,\ new]$ processed the update that node $s[k, new]$ sent at time $t_{s[k+1,\ new]}$, then $D_{j\ s[k,\ new]}^{s[k-1,\ new]}(t) = D_j^{s[k,\ new]}(t) = D_{j\ s[k+1,\ new]}^{s[k,new]}(t) + d_{s[k,new]s[k+1,\ new]}(t) > D_{j\ s[k+1,\ new]}^{s[k,new]}(t)$. However, if $s[k-1,\ new]$ did not process the update that node $s[k,\ new]$ sent at time $t_{s[k+1,\ new]}$, then $D_{j\ s[k,\ new]}^{s[k-1,\ new]}(t) = D_j^{s[k,\ new]}(t_{s[k+1,\ old]}) > D_{j\ s[k+1,\ new]}^{s[k,\ new]}(t)$, because router $s[k+1,\ new]$ must be a feasible successor for router $s[k, new]$ to make it its successor at time $t_{s[k+1,\ new]}$. Therefore, if router $s[k,\ new]$ is already passive when it changes successor at time $t_{s[k+1,\ new]}$, then $D_{j\ s[k,\ new]}^{s[k-1,\ new]}(t) > D_{j\ s[k+1,\ new]}^{s[k,\ new]}(t)$

Alternatively, consider the case in which router $s[k,\ new]$ is active from time $t_k < t$ to time $t_{s[k+1,\ new]}$ when it becomes passive again to join $P_{aj}(t)$. In this case, regardless of the value of $D_j^{s[k,\ new]}(t_k)$, router $s[k,\ new]$ must have sent a query to its neighbors with $RD_j^{s[k,\ new]}(t_k) = \infty$ at time t_k, and all of those neighbors must acknowledge that value of $RD_j^{s[k,\ new]}(t_k)$ before router $s[k,\ new]$ can make any changes to its distance at time $t_{s[k+1,\ new]}$.

When router $s[k-1,\ new]$ makes router $s[k,\ new]$ its successor when it joins $P_{aj}(t)$ at time $t_{s[k,\ new]} \leq t$, it may or may not have processed any update or query sent by node $s[k,\ new]$ at time $t_{s[k+1,\ new]} \leq t$ when that node joins $P_{aj}(t)$. In the first case,

$$D_{j\ s[k,\ new]}^{s[k-1,\ new]}(t) = RD_j^{s[k,\ new]}(t_{s[k+1,\ new]}) = D_j^{s[k,\ new]}(t_{s[k+1,\ new]})$$

$= D_j^{s[k,\ new]}(t) \geq FD_j^{s[k,\ new]}(t) > D_{j\ s[k+1,\ new]}^{s[k,\ new]}(t)$. In the second case, $D_{j\ s[k,\ new]}^{s[k-1,\ new]}(t) = RD_j^{s[k,\ new]}(t_k)$; this is impossible, because $RD_j^{s[k,\ new]}(t_k) = \infty$ and node $s[k-1,\ new]$ could not have chosen a neighbor reporting an infinite distance as its successor.

From the above argument it follows that if a router $s[k, new]$ is passive at time t, then $D_{js[k,new]}^{s[k-1,new]} > D_{js[k+1,new]}^{s[k,new]}(t)$. However, because all routers in the loop $C_j(t)$ are passive at time t, traversing path $P_{ai}(t)$ leads to the erroneous conclusion $D_{ja}^i(t) > D_{ja}^i(t)$. This implies that a loop cannot be formed when $S_j(G)$ is loop free before time t and G has a stable topology. On the other hand, the handling of queries and replies in LPA is not modified with the establishment or failure of links. Therefore, the theorem is true.

Q.E.D.

4.4 Correctness of LPA

To prove that LPA converges to correct routing-table values in a finite time, we assume that there is a finite time T_c after which no more link-cost or topology changes occur.

Lemma 4 LPA is free of deadlocks.

Consider the case in which the network has a stable topology. When a router is in the active state and receives a query from a neighbor, the router replies to the query with an infinite distance. A router updates its distance table entries when either an update or a reply message is received in active state. On the other hand, when a router in passive state receives a query from its neighbor, it computes the feasible distance and updates its distance and routing tables accordingly. If a router finds a feasible successor, it replies to its neighbor's query with its current distance to the destination. If a router can find no feasible successor, it forwards the query to the rest of is neighbors and sends a reply with an infinite distance to the neighbor who originated the query. Accordingly, in a stable topology, a router that receives a query from a neighbor for any destination must answer with a reply within a finite time, which means that any router that sends a query in a stable topology must become passive after a finite time.

Consider now the case in which the network topology changes. When a link fails or is reestablished, an active router that detects the link status change simply assumes that a router at the other end of the link has reported an infinite distance and has replied to the ongoing query. Because an active router must detect the failure or establishment of a link within a finite time, and because router failures or additions are treated as multiple link failures or additions, it follows from the previous case that no router can be active for an indefinite period of time and the lemma is true.

Q.E.D.

Lemma 4.1 *TRT is correct.*

Proof: TRT is correct if the tag value given by TRT at router i for destination j equals correct. This is true only when the neighbor n that router i chooses as successor to j offers the smallest distance from i to each node in its reported implied path from n to j.

First note that, procedure DT is executed before TRT and ensures that router i sets $D^i_{jb} = \infty$ if its neighbor b reports a path to b that includes i. Therefore, TRT deals with simple paths only.

According to procedure TRT, there are two cases in which a router stops tracing the routing table (a) the trace reaches node i itself (*i.e,* $p^i_{xns} = i$), and (b) a node on the path to j is found with tag^i_x = correct. We prove that the correct path information is reached in both cases.

Case 1: Assume that TRT is executed for destination j after an input event. The tag for each destination affected by the input event is set to null before procedure TRT is executed. Therefore, if TRT is executed for destination j and node i (the source) is reached, the tag of each node in the path from i to j through neighbor n must be null. Therefore, the distance from i to j through n is the shortest path among all neighbors since node i chooses the minimum in row entry among its neighbors for a given destination j. The lemma is true for this case.

Case 2: If node x_1 with $tag^i_{x_1}$ = correct is reached, then it must be true that either node i or a node x_2 with $tag^i_{x_2}$ = correct is reached from x_1.

If node i is reached from x_1, then it follows from case 1 that neighbor n offers the smallest distance among all of i's neighbors to each node in the implied subpath from n to x_1 reported by neighbor n. Furthermore, because x_1 is reached from j, node n must also offer the smallest distance among all of i's neighbors to each node in the implied subpath from x_1 to j reported by n. Therefore, it follows that the lemma is true if node i is reached from x_1 (from case 1). Otherwise, if x_2 is reached, the argument used when i is reached from x_1 can be applied to x_2. Because router i always sets tag^i_i = correct and TRT deals with simple paths only, this argument can be applied recursively only for a maximum of $h < \infty$ times until i is reached, where h is the number of hops in the implicit path from n to j reported by n to i. Therefore, case 2 must eventually reduce to case 1 and it follows that the lemma is true.

Lemma 5 The change in the cost or status of a link will be reflected in the distance and the routing tables of a router adjacent to the link within a finite time.

Proof: Regardless of the state in which router i is for a given destination j, it updates its link-cost and distance table within a finite time after it is notified of an adjacent link changing its cost, failing, or starting up. On the other hand, router i is allowed to update its routing table for destination j only when it is in passive state for that destination. However, because LPA is live (Lemma 4), if router i is active for destination j, it must receive all the replies to its query regarding j within a finite time, i.e., when it becomes passive. When router i becomes passive for destination j, it executes Procedure TRT, which updates the routing-table entry for destination j using the most recent information in router i's distance table (Lemma 4.1). This implies that any change in a link is reflected in the distance and routing tables of a neighbor router within a finite time T.

Q.E.D.

Given Lemma 5 and our assumption about time T_c, a finite time must exist when all routers adjacent to the links that changed cost or status have updated their link cost and status information, and after which no more link-cost or topology changes occur. Let T denote that time, where $T_c \leq T < \infty$.

Theorem 6 *After a finite time $t \geq T$, the routing tables of all routers must define the final shortest path to each destination.*

Proof: Let $T(H)$ be the time at which all messages sent by routers with shortest paths having $H-1$ hops ($H \geq 1$) to a given destination j have been processed by their neighbors.

Assume that destination j is reachable from every router.

For any router a adjacent to j, it follows from Lemma 5 that, if router a's shortest path to j is the link (a, j), then router a must update $D_j^a = d_{aj}$ by time $T = T(0)$ and the theorem is true for $H = 0$.

Because LPA is loop free at every instant (Theorem 5), the number of hops in any shortest path (as implied by the successor entries for destination j in all the routing tables) is finite. Accordingly, the proof can proceed by induction on H.

Assume that the theorem is true for some $H > 0$. According to this inductive assumption, by time $T(H)$, router i must have a correct routing-table entry for every destination for which it has a shortest path of H hops or less. Property 4 must be satisfied for all such destinations. On the other hand, from the definition of $T(H+1)$, it follows that any update messages sent by routers with shortest paths of H hops or less to j or any other destination have been processed by their neighbors by time $T(H+1)$. Therefore, if router i's shortest path to destination j has $H+1$ hops, Property 4 must be satisfied at router i for that destination by time $T(H+1)$, because all possible predecessors for destination j must satisfy Property 4 at router i and that router must have the correct information for link (i, s^i_j) at time $T(0) < T(H+1)$ (Lemma 5). It follows that the theorem is true for the case of a connected network.

Consider the case in which j is not accessible to a connected component C of the network. Assume that there is a router $i \in C$ such that $D^i_j < \infty$ at some arbitrarily long time. If that is the case, j must satisfy Property 4 through at least one of router i's neighbors at that time; the same applies to such a neighbor, and to all routers in at least one path from i to j defined by the routing tables of routers in C. This is not possible, because C is finite and LPA is always free of loops and live, which implies that, after a finite time $t_f \geq T$, all paths to j defined by the successor entries in the routing tables of routers in C must lead to routers that have set their distance to j equal to ∞. Therefore, because C is finite, LPA is live, and messages take a finite time to be transmitted, it follows that destination j will fail to satisfy Property 4 at each router within a finite time $t \geq t_f$, who must then set its distance to infinity, and the theorem is true.

Q.E.D.

Theorem 7 *A finite time after t, no new update messages are being transmitted or processed by routers in G, and all entries in distance and routing tables are correct.*

Proof: After time T, the only way in which a router can send an update message is after processing an update message from a neighbor. Accordingly, the proof needs to consider three cases, namely: router i receives an update, a query, or a reply from a neighbor.

Consider an arbitrary router $i \in G$. Because LPA is live (Theorem 5) and router i obtains its shortest distance and corresponding path information for destination j in a finite time after T (Theorem 6), router i must be passive within a finite time $t_i \geq T$.

If router i receives an update for destination j from router k after time t_i, router i must execute Procedure Update. If router i has no path to destination j, D^i_j must be infinity and router k must report an infinite distance as well, because router i achieves its final shortest-path at time t_i; in this case, router i simply updates its distance table. On the other hand, if router i has a path to destination j, then $D^i_j < \infty$ and router i must find that FC is satisfied and execute Procedure TRT. Because an update entry is added only when the shortest distance or predecessor to j change, router i can send no update or query of its own.

If router i receives a query from a neighbor for destination j after time t_i, it must execute Procedure Query. If router i has no physical path to destination j, D^i_j must be infinity and router k must report an infinite distance in its query, because router i achieves its final shortest-path at time t_i; in this case, router i simply updates its distance table and sends a reply to router k with an infinite distance. On the other hand, if router i has a physical path to destination j, it must determine that FC is satisfied when it processes router k's query. Accordingly, it simply sends a reply to its neighbor with its current distance and predecessor to router j. Therefore, router i cannot send an update or query of its own when it processes a query from a neighbor after time t_i.

After time t_i, router i cannot receive a reply from a neighbor, unless it first sends a query after time t_i, which is impossible according to the above two paragraphs.

It follows from the above that, for any given destination, no router in G can generate a new update or query after it reaches its final shortest path and predecessor to that destination. Because every router must obtain its final shortest distance and predecessor to every destination within a finite time (Theorem 6), the theorem is true.

Q.E.D.

4.5 Complexity of LPA

This section compares LPA's worst-case performance with respect to the performance of DBF, DUAL, and ILS. This comparison is made in terms of the overhead required to obtain correct routing-table entries a assuming that the algorithm behaves synchronously, so that every router in the network executes a step of the algorithm simultaneously at fixed points in time. At each step, router receives and processes all the inputs originated during the preceding step and if required, sends update messages to the neighboring routers at the same step. The first step occurs when at least one router detects a topological change and issues update messages to its neighbors. During the last step, at least one router receives and processes messages from its neighbors and after which router stops transmitting any update messages till a new topological change has taken place. The number of steps taken for this process is called the *time complexity* (TC); the number of messages required to accomplish this is called the *communication complexity* (CC).

DBF has a worst-case time complexity of $O(|N|)$ and worst-case communication complexity of $O(|N^2|)$, where $|N|$ is the number of routers in the network G [GLA92]. ILS requires that each change in the cost or status of a link be communicated to all routers in the network; accordingly, it has $TC = O(d)$ (where d is the network diameter), because a link-state update must traverse the whole network, and $CC = O(E)$, because each update traverses each link at most once in ILS but each link has two states, one in each direction of the link. On the other hand, DUAL has $TC = O(x)$ and $CC = O(x)$, where x is the number of routers affected by the single topology change [GLA92]. The following theorem shows that LPA has $TC = O(x)$; using a similar argument, LPA can be shown to have a worst-case communication complexity of $O(x)$ after a single resource failure.

Theorem 8 *The time complexity for a single link failure or link-cost change of LPA is $O(x)$ in the worst-case, where x is the number of routers affected by the change.*

Proof: Let the source router be i, destination router be j and the failed link be (n, m) where router m is downstream to router n.

Here also we have the same four cases as in Theorem 4.

A router with the initial shortest path not going through the changed link (Case 1) does not change its routing table, because the original shortest path is not changed and the change in the link cost has only resulted in the increase in path length through other routes.

In Case 2, router i will be aware of the change in the link cost along the shortest path after a delay not exceeding the number of links on the new shortest-path. In Case 3, the change will be noticed in the worst-case after a delay of at most the number of links affected by the link cost change.

Let router n_k with the original shortest path through the changed link be k hops away from router n on the initial shortest path. When a link cost changes or the link fails, the router containing the failed link selects a new neighbor for a path to the destination j, if the successor satisfies the feasibility condition. If a feasible successor is not found, router will send queries to its neighbors and sends a reply and query to the originator of the query. Queries will propagate down the routing tree which is affected by the link cost change. The feasible distance will be eventually determined after a worst-case delay of the number of links affected by the link-cost change. Thus all routers involved in the path will have correct path information. Once the feasible distance has been found, the routing table entries are updated and the appropriate update messages will be sent to the neighboring routers. The distance of the stable router found in the path from i to j in the new shortest path is bounded by x, the number of routers affected in the shortest path. Therefore, in the worst-case, the number of steps required for a router to converge to its correct distance is $O(x)$.

Q.E.D.

4.6 Summary

In this chapter, we have presented and verified the first routing algorithm (LPA) that eliminates the formation of temporary routing loops without inter-nodal synchronization mechanism spanning multiple hops or the communication of complete or variable length path information. LPA is based on the notion of using information about the second to last hop (or predecessor) of shortest paths to ensure termination, and an efficient inter-neighbor coordination mechanism to eliminate temporary loops. Detailed proofs of loop-freedom and correctness of LPA were presented and LPA's complexity was analyzed. The worst-case complexity of LPA for a single recovery or failure is $O(x)$, with x being the number of nodes affected by this recovery or failure.

Chapter 5
Simulation

In this Chapter, we present the simulation results for PFA and LPA, which were described and verified in Chapters 3 and 4. The performance of these two algorithms is compared with that of DUAL [GLA93] and an ideal link state (ILS) algorithm. DUAL and ILS were chosen, because they represent the state-of-the-art in internet routing protocols.

The rest of the Chapter is organized as follows. The next section gives a brief introduction about the simulator that has been used in our simulation experiments. Section 5.2 explains the design of our simulator. Section 5.3 list the parameters used in measuring the performance and the instrumentation part of the simulations. Section 5.4 describes the results of the simulations. Finally, Section 5.5 summarizes the chapter.

5.1 Simulations in Drama

We have developed our simulations using an actor-based, discrete-event simulation language called *Drama* [Zau91] together with a network simulation library. Drama is a C-based simulation language that supports an actor-based computational model, in which actors are independent activities that communicate by passing messages in the context of a discrete-event simulation. This makes it convenient for modeling communication networks.

Drama contains a library of functions, some of which look up user-supplied functions from tables. The language extensions are mostly declarative. The system contains a translator and a corresponding run-time library. The debugging support for the runtime library gives a pseudocode description of each function instead of the source code itself. Drama simulations deal with several classes of objects which include *simulations* and *actors* as the primary classes. The user declares pointers to these objects, and Drama primitives are used to create them. A *script* must be supplied to create an actor. Scripts are the functions that program an actor's behavior.

Communication among actors is done by means of message passing. Messages are posted on simulation's event queues. There can be two types of variables in Drama – *instant variable* and *static variable.* Instant variable are similar to static variables in C whereas each actor has one copy of the static variable. To customize simulations, Drama provides *daemons*, *hooks*, *buffers* and *handlers.* To each of the objects in Drama (actor, simulations and queues), a buffer or a daemon can be attached. For simulations, the daemon can be called each time the event queue is checked for the next event. Thus, it is easy to have a simulation run for a user-defined time.

5.2 Design of the Simulator

The network simulation library of Drama treats both nodes and links in the networks as *actors.* Nodes send packets over the links using the functional-call interface to the link's actor. The packets are received by responding to the messages delivered from the event queue. Link failures and recoveries are handled by sending a link-status message to all nodes at the end points of the appropriate link. In the link model used in the simulation, link propagation time is an input parameter which can be changed during the course of simulation. We have modeled all runs with unit propagation time. If a link fails, all the packets in transit are dropped.

The simulation of the algorithms is based on the pseudocode description of algorithm given in the previous chapters. In our simulations, a node responds to the receipt of a message by running the routing algorithm and sending the required updates to its neighboring nodes. Outgoing messages are queued at a node after waiting for some processing time. If any incoming packets arrive before the processing time expires, the routing algorithm is run again and the new packets that are generated are queued. Once the processing time for all the events expire, depending on the algorithm, the redundant updates are removed and the queues are sent over the links. In our runs, we have set the processing time to zero. The internal mechanism of Drama ensures that all updates due to arrive at the current simulation time are processed before the generation of new updates. Due to this mechanism, in the simulations, multiple updates are put into the same packet. This makes the number of packets that are being transmitted an important performance measure than the number of bits that are actually transmitted.

We have simulated PFA and LPA together with an ILS using Dijkstra's shortest-path algorithm and DUAL. In the case of PFA and LPA, a routing table containing the *predecessor* and *successor* information were generated. Simulations of both algorithms are based on an *incremental update* mechanism.

5.3 Parameters

We have instrumented the simulations in two ways. The simplest way is to have a set of counters that can be reset at various points and are updated appropriately. These counters determine the statistics such as the total number of messages sent, total time taken etc. at all nodes. The value of these counters are recorded when the event queue becomes empty (which implies the algorithm has converged). These counters themselves are of two types. They can be associated with the individual nodes and links or can be associated with all nodes and links.

Statistics are also collected after the processing of each event. Drama supports this by means of calling a function whose convention is specified by the run-time library. The routing tables are characterized after each step thereby allowing us to characterize the routes produced by the algorithm. Each input event received during the same simulation step are processed independently.

After each link or node failure or recovery, or change in the cost of a link, the algorithm is allowed to run to convergence. Node failures are modeled as all the links connected to that node going down simultaneously, and node recovery is modeled as all the links that connect to that node coming back up simultaneously. The quantities that were measured during the simulation include

Events: The total number of updates (including queries and replies in LPA and DUAL) and the changes in the link costs processed by nodes.

Packets: The total number of packets transmitted over the network. Each of these packets may contain multiple updates.

Duration: The total time elapsed for the algorithm to converge.

Operations: The total number of operations performed by the algorithm. The operation count is incremented when an event occurs (and whenever procedure *rt_update* is called in case of PFA).

Both the mean and the standard deviation of the above measures are computed.

5.4 Results

To obtain insight on the average performance of PFA and LPA in a real network, simulations were run using the topologies of typical networks after we performed a series of tests on smaller topologies for debugging purposes. The main network topologies considered are LOS-NETTOS, NSFNET-T1-Backbone, and ARPANET as shown in Figure 5.1. We selected these topologies to compare the performance of the routing algorithms for the well-known cases, given that we cannot sample a large enough number of networks. We have performed a comprehensive number of tests on these topologies. For each network, we generated test cases consisting of all single failures and recoveries both for links and nodes. The routing algorithm was allowed to converge after each such change. The link costs are always positive and greater than zero. *Infinite cost* is used to represent a failed link. In our simulations, we have modeled each link to be of unit cost. However, this can be easily changed to any number greater than zero or can be associated with a cost metric.

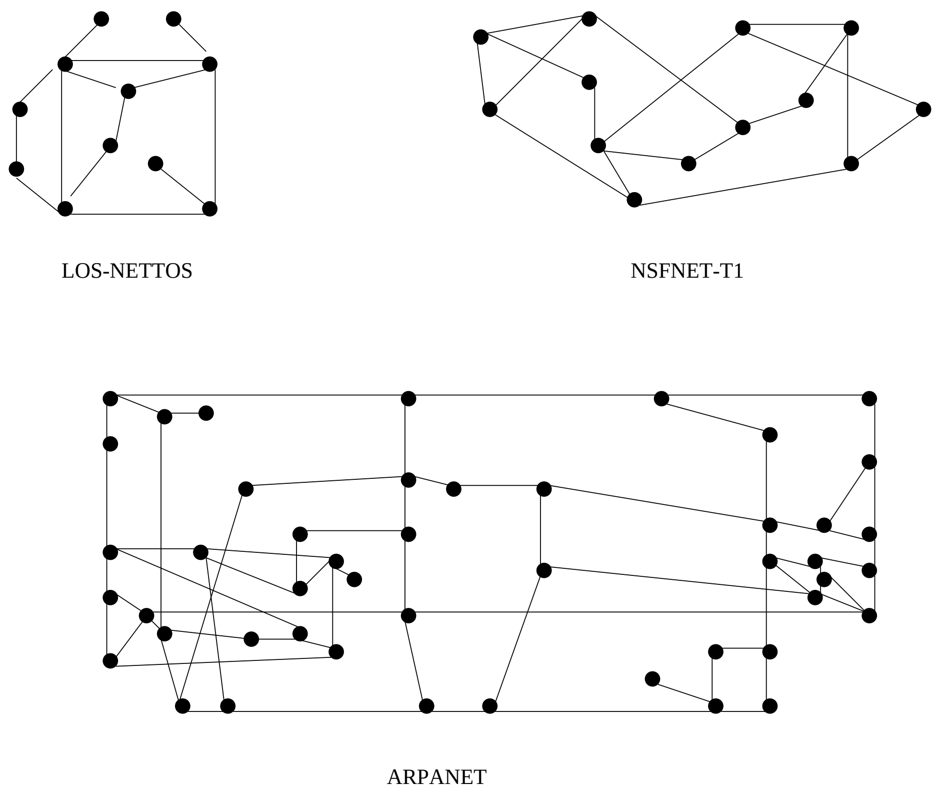

Figure 5.1: Network Topologies

In all cases, nodes were assumed to perform computations in zero time and links were assumed to provide one time unit of processing delay. The link model allows link delay and link cost to be set independently. The simulation uses link weights of equal cost (unit cost). Each unit of time therefore represents a step in which all currently available packets are processed. Even though the simulation proceeds synchronously, node model allows the packets to be processed asynchronously. Thus, each event at a node is processed independently of other events received during the same simulation step. During each simulation step, a node processes input events received during the previous step one at a time, and generates messages needed for each input event it processes. To obtain the average figures, the simulation makes each link (node) in the network fail, and counts the steps and messages needed for each algorithm to recover. It them makes the same link (node) recover and repeats the process. The average is taken over all the link (node) failures and recoveries. The results of this simulation are shown in Tables 5.1–5.4. These results are compared with that of DUAL and an ideal link-state algorithm (ILS) running Dijkstra's shortest-path algorithm.

For a single resource failure or recovery, the operations count for the Dijkstra's link state algorithm is substantially higher than that for other algorithms, often more than a magnitude higher. This is expected because ILS forces a router to recompute its shortest paths using the new topology.

PFA, LPA and DUAL have better overall average performance than ILS after the recovery of a single router or a link. This is also expected of any efficient distance vector algorithm, because routers propagate updates only when they change their routing tables, while ILS floods the entire network with the same link-state update. The performance of PFA and LPA are comparable to ILS after the failure of a single router or a link. This is a remarkable improvement over DUAL, which requires approximately twice the number of steps to converge than ILS after failures. Insofar as overhead traffic is concerned, PFA and LPA are comparable to DUAL and ILS. PFA and LPA converge faster than DUAL and is more responsive than DUAL.

The time required for PFA to converge is half that of DUAL. The number of packets (messages) exchanged among nodes is also more than 50% less than that of DUAL. However, the event count and the operation count is about 2 to 3 times higher than DUAL. The convergence time if PFA is better compared to ILS for larger networks.

The convergence time for LPA is better than that of DUAL for single resource failure or recovery. Also, the number of packets exchanged after each step is upto one third that of DUAL for large networks and the event count is comparable to DUAL. As the size and the connectivity of the network increases, LPA performs better compared to DUAL. Compared to ILS, the results obtained for LPA are very encouraging. A minimum overhead of two or three steps over the number of steps needed to traverse the network along the fastest path is needed to handle queries. Therefore, our results indicate that not considering this overhead, LPA tends to update routing tables as fast as it can be done with ILS. The results obtained for PFA for link or node failures also support this conclusion. In general, however, PFA and

Table 5.1: Routing Algorithm Response to a Single Link Failure

Parameter	PFA		LPA		DUAL		ILS	
	mean	sdev	mean	sdev	mean	sdev	mean	sdev
Los-Nettos Link-Failure Cases								
Event Count	45.7	17.9	94.7	33.8	49.9	18.6	29.0	5.8
Packet Count	13.5	6.01	26.3	6.45	32.6	11.8	27.0	5.8
Duration	2.86	0.74	4.09	0.79	6.7	1.33	4.2	0.88
Operation Count	62.4	18.03	67.4	16.9	69.9	18.6	724.1	27.3
NSFNET Link-Failure Cases								
Event Count	105.9	55.21	160.2	63.7	91.1	46.2	53.0	0.0
Packet Count	28.7	12.7	46.22	13.34	53.7	18.5	51.0	0.0
Duration	3.7	0.79	5.56	1.06	6.9	0.88	5.3	0.25
Operation Count	113.1	51.8	106.1	31.8	118.1	46.2	1840.1	16.4
ARPANET Link-Failure Cases								
Event Count	962.1	392.9	587.3	381.5	720.9	449.1	160.0	0.0
Packet Count	96.5	45.9	126.1	59.8	266.8	97.3	158.0	0.0
Duration	7.16	1.75	9.24	3.39	15.1	3.45	8.5	0.74
Operation Count	843.90	594.5	385.6	190.8	813.9	449.1	25600.2	57.121

Table 5.2: Routing Algorithm Response to a Single Link Recovery

Parameter	PFA		LPA		DUAL		ILS	
	mean	sdev	mean	sdev	mean	sdev	mean	sdev
Los-Nettos Link-Recovery Cases								
Event Count	91.3	15.5	64.0	11.3	45.7	7.45	33.3	1.86
Packet Count	18.0	5.04	10.36	2.06	17.0	7.25	31.9	1.86
Duration	2.93	0.46	3.27	0.62	3.71	0.88	3.86	0.46
Operation Count	109.6	24.6	52.0	5.65	65.7	7.45	944.0	45.8
NSFNET Link-Recovery Cases								
Event Count	162.8	45.9	98.8	24.4	67.5	18.8	56.9	2.39
Packet Count	36.3	12.4	16.0	3.38	22.0	6.15	54.9	2.39
Duration	3.4	0.49	4.17	0.5	3.86	0.4	4.7	0.4
Operation Count	171.1	50.4	75.4	12.2	93.7	18.8	2140.4	80.6
ARPANET Link-Recovery Cases								
Event Count	638.2	370.3	242.4	112.8	362.2	147.6	162.7	15.4
Packet Count	108.6	48.9	33.0	25.5	79.3	21.3	160.7	15.4
Duration	6.89	1.51	5.96	2.75	7.3	1.46	7.84	0.67
Operation Count	1144.9	620.1	213.2	56.4	454.2	147.6	26900.8	2477.9

Table 5.3: Routing Algorithm Response to a Single Node Failure

Parameter	PFA		LPA		DUAL		ILS	
	mean	sdev	mean	sdev	mean	sdev	mean	sdev
Los-Nettos Node-Failure Cases								
Event Count	135.6	79.5	88.18	27.42	73.0	25.4	31.8	9.6
Packet Count	39.8	17.5	25.72	6.53	45.5	3.26	26.7	7.19
Duration	5.82	2.85	4.36	1.07	6.91	0.99	4.09	0.51
Operation Count	195.0	112.2	95.0	36.65	123.9	50.2	702.9	204.3
NSFNET Node-Failure Cases								
Event Count	176.4	48.8	160.7	64.3	176.6	78.1	64.9	8.17
Packet Count	46.8	12.7	41.42	9.45	97.2	23.7	58.7	7.33
Duration	4.8	0.98	4.93	0.96	12.6	5.13	5.21	0.3
Operation Count	262.6	68.4	158.4	42.7	253.2	89.1	2070.9	234.6
ARPANET Node-Failure Cases								
Event Count	1350.8	373.8	646.5	424.4	1050.4	300.8	218.8	67.1
Packet Count	96.6	75.9	144.7	55.3	382.6	81.2	212.1	65.1
Duration	5.4	3.4	9.12	2.4	17.8	9.2	8.6	0.72
Operation Count	1803.8	407.4	589.5	271.3	1320.8	563.5	33356.7	10766.2

Table 5.4: Routing Algorithm Response to a Single Node Recovery

Parameter	PFA		LPA		DUAL		ILS	
	mean	sdev	mean	sdev	mean	sdev	mean	sdev
Los-Nettos Node-Recovery Cases								
Event Count	221.1	117.9	105.6	67.42	94.3	40.5	56.2	13.4
Packet Count	30.4	10.3	13.09	7.01	41.0	12.4	51.1	10.8
Duration	3.18	0.38	3.09	0.89	4.7	0.44	4.4	0.5
Operation Count	274.9	136.4	103.7	59.6	145.2	66.5	1698.8	478.4
NSFNET Node-Recovery Cases								
Event Count	379.2	94.6	177.4	67.9	154.2	36.9	92.8	4.6
Packet Count	51.4	9.7	22.6	9.13	60.2	9.3	86.6	3.6
Duration	3.8	0.4	4.14	1.5	4.6	0.49	5.9	0.3
Operation Count	486.0	137.2	166.7	48.7	234.8	49.8	4150.8	239.8
ARPANET Node-Recovery Cases								
Event Count	980.4	699.7	551.6	296.4	691.9	235.5	301.2	45.3
Packet Count	107.2	80.1	68.06	42.03	207.9	46.7	294.5	42.9
Duration	5.27	2.56	7.78	3.33	8.5	0.73	9.6	1.14
Operation Count	3252.0	1911.5	542.0	224.4	957.6	347.3	50102.2	7930.4

LPA cannot guarantee that up-to-date information is not overwritten by stale information (which is eventually corrected). This opens up a research question of how to ensure that a router updates its distance and routing table using only more recent distance vectors.

Node failure is the worst case in DUAL's performance. PFA and LPA's performance is comparable to ILS even for node failure and recovery cases and performs better than DUAL in terms of convergence time and the number of packets exchanged. The number of operations performed in PFA for single router or link failure or recovery is almost twice that of DUAL. However, it is an order of magnitude less than that of ILS. The number of operations performed in LPA is always less than DUAL.

The comparison between the performance of PFA and LPA clearly indicates that the inter-neighbor synchronization mechanism of LPA does not introduce excessive overhead on the algorithm's performance and the faster convergence time of LPA as compared to DUAL is due to the fact that LPA achieves loop freedom by blocking potential temporary loops (procedures DT and TRT) using a single-hop inter-neighbor synchronization mechanism. In contrast, DUAL uses queries that involve many nodes in the network and as many hops as the worst-case hop length of a path. PFA incurs fewer steps than the rest of the algorithms after single failures. This is because of procedure DT, which prevents the formation of temporary loops without internodal coordination. However, the results obtained for LPA after router or link failures are very encouraging. LPA is better than PFA in terms of the number of messages exchanged for resource recoveries and the number of operations performed after any type of resource change. This is due to Procedure TRT and the use of *tags* which eliminate the need to traverse the complete path from destination to the source after each input event is processed. Because of the inter-neighbor synchronization scheme used in LPA, it can be expected that at least two additional steps are required to converge, in addition to the steps required to propagate updates across the network.

The above results indicate that LPA constitutes a more scalable solution for routing in large internets than ILS and even DUAL. In fact, LPA constitutes the most efficient distance vector algorithm reported to date. After resource failure, LPA incurs similar number of steps and overhead traffic as ILS, but requires much fewer operations at each router. After resource addition, LPA requires fewer steps, messages and operations than DUAL and ILS. Because of the inter-neighbor synchronization needed in LPA to ensure loop-freedom, PFA outperforms LPA on some instances (in terms of the number of steps and messages); however, LPA always requires fewer operations and is always free of loops.

5.5 Summary

In this chapter, we have discussed the simulation results for PFA and LPA and have compared them with DUAL and an ILS that uses Dijkstra's shortest-path algorithm. Simulations have been carried out using a C-based simulation language called Drama along with the network simulation library. The results indicate that the two proposed algorithms converge faster than DUAL and ILS, while exchanging fewer messages for single resource changes. The comparison of PFA and LPA indicates that LPA achieves loop-freedom without excessive additional overhead. LPA is clearly the best alternative among the algorithms simulated.

Chapter 6
Conclusions

Results: Path-finding algorithms are an attractive alternative to DBF for distributed routing, because they eliminate counting-to-infinity problem. However, current path-finding algorithms can incur substantial temporary loops in the paths specified by predecessor information before they converge, which leads to slower convergence.

In this thesis, we have proposed two new algorithms to the class of path-finding algorithms that eliminates the formation of temporary loops. We introduce, verify and analyze these two algorithms, which we refer to as PFA and LPA. Both of these algorithms operate by specifying the second-to-last-hop to each known destination, along with the distance to the destination. Unlike earlier algorithms, PFA and LPA, upon receiving an update from its neighbor k, determines if a path to destination through any of its other neighboring nodes includes neighbor k itself. This step reduces the possibility of temporary loops. LPA achieves loop-freedom at every instant using the implicit path information and an inter-neighbor coordination mechanism that spans over single hop only.

The proposed algorithms use the same amount of space as the basic path-finding algorithms. The performance of these two algorithms has been compared with that of DUAL and an algorithm based on ideal link-state algorithm which constitute the state of the art in the present-day internet routing. The simulations were carried out using a C-based simulation language Drama along with a network simulation library. The results indicate that the two proposed algorithms converge faster than DUAL and ILS, while exchanging fewer messages for single resource changes. The comparison of LPA and PFA indicates that LPA achieves loop-freedom without excessive additional overhead.

Our simulation results show that LPA converges faster than DUAL for single-resource changes and that the number of messages exchanged is comparable to the number obtained for DUAL. LPA is comparable to ILS insofar as number of steps and number of messages needed for convergence after resource failures, and is faster than ILS after resource recoveries and requires fewer operations than ILS. Taking the average number of steps, messages and operations into account, the results indicate that LPA constitutes a more scalable solution for routing than ILS or even DUAL.

Future Work: Our research indicates that LPA is the most efficient loop-free routing algorithm reported to date and, perhaps, the most efficient distributed shortest-path routing algorithm. A research problem that neeeds to be investigated is how to ensure that routers using LPA will never update their routing information using stale distance vectors. One of the other research interests could be to simulate data traffic and measure the resulting packet loss from looping.

Because of the growing size of internetworks, it would be of great interest to extend LPA to hierarchical networks. A promising approach to address this problem is to adopt McQuillan's scheme to hierarchical routing. Our simulation results for LPA and ILS seem to indicate that this new hierarchical routing scheme should outperform OSPF, because the latter is based on topology broadcast algorithms. Comparing the performance of both schemes constitutes another research question.

Bibliography

[BG92] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992. Second Edition.

[Ceg75] T. Cegrell. A Routing Procedure for the TIDAS Message Switching Network. *IEEE Trans. Commun.*, (23(6)):575–585, 1975.

[CKGLA92] C. Cheng, S. Kumar, and J.J. Garcia-Luna-Aceves. An Efficient Loop-Free Algorithm. submitted for publication, 1992.

[CRKGLA89] C. Cheng, R. Reley, S.P.R. Kumar, and J.J. Garcia-Luna-Aceves. A Loop-Free Extended Bellman-Ford Routing Protocol without Bouncing Effect. In *Computer Communications Review*, pages 224–236. ACM, 19 (4) 1989.

[GLA86] J.J. Garcia-Luna-Aceves. A Fail-Safe Routing Algorithm for Multihop Packet-Radio Networks. In *INFOCOM.* IEEE, Miami, Florida, April 1986.

[GLA89] J.J. Garcia-Luna-Aceves. A Minimum-Hop Routing Algorithm based on Distributed Information. *Computer Networks and ISDN Systems*, (Vol. 16):367–382, 1989.

[GLA92] J.J. Garcia-Luna-Aceves. Distributed Routing with Labeled Distances. In *INFOCOM*, volume 2, pages 633–643. IEEE, 1992.

[GLA93] J.J. Garcia-Luna-Aceves. Loop-Free Routing using Diffusing Computations. *IEEE/ACM Trans. Networking*, 1(1):130–141, Feb. 1993.

[Hag83] J. Hagouel. Issues in Routing for Large and Dynamic Networks. Technical Report RC 9942 (No. 44055), IBM Research Report, April 1983.

[Hed88] C. Hedrick. Routing Information Protocol. RFC 1058, June 1988.

[HS82] R. Hinden and A. Sheltzer. DARPA Internet Gateway. RFC 823, Network Information Center, SRI International, September 1982.

[Hum91] P.A. Humblet. Another Adaptive Shortest-Path Algorithm. *IEEE Trans. Commun.*, (Vol.39, No.6):995–1003, June 1991.

[JM82] J.M. Jaffe and F.M. Moss. A Responsive Routing Algorithm for Computer Networks. *IEEE Trans. Commun.*, (Vol. 30):1758–1762, July 1982.

[McQ74] J. McQuillan. Adaptive Routing algorithms for Distributed Computer Networks. Technical report, Bolt Beranek and Newman, BBN, Report.2831, 1974.

[Mil83a] D. Mills. DCN Local Network Protocols. RFC 891, Network Information Center, SRI International, December, 1983.

[Mil83b] D. Mills. Exterior Gateway Protocol. RFC 904, Network Information Center, SRI International, December 1983.

[Moy91] J. Moy. OSPF Version 2. RFC 1247, August 1991.

[Ora90] D. Oran. OSI IS-IS Intra-domain routing protocol. RFC 1142, February 1990.

[Per91] Radia Perlman. A comparison between two routing protocols: OSPF and IS-IS. *IEEE Network*, (Vol. 5, No. 5):18–24, Sept. 1991.

[RF91] B. Rajagopalan and M. Faiman. A Responsive Distributed Shortest-Path Routing Algorithm within Autonomous Systems. *Internetworking: Research and Experience*, (Vol. 2, No. 1):51–69, March 1991.

[RL94] Y. Rekhter and T. Li. Border Gateway Protocol 4 (BGP-4). Network Working Group Internet Draft, January 1994.

[Sch86] M. Schwartz. *Telecommunications Networks: Protocols, Modeling and Analysis.* Addison-Wesley Publishing Co., Menlo Park, California, 1986. Chapter 6.

[Zau91] W.T. Zaumen. Simulations in Drama. Network Information System Center, SRI International, January 1991.